Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Synthetic Infrared Scene

Improving the KARMA IRSG module and signature modelling tool SMAT

Marc-André Labrie
Eric Rouleau
Jonathan Richard
Mathieu Desmeules
Alexandre Bastien
Louis Tanguay Informatique inc.

Geoffroy Rivet-Sabourin
Technologie Intelligence Image inc.

Prepared By:

Louis Tanguay Informatique inc.
825 Boulevard Lebourgneuf, Bureau 204
Québec, Canada G2J 0B9

Contractor's Document Number: LTI-SIS-2011-1
Contract Project Manager: Marc-André Labrie
PWGSC Contract Number: W7701-082234/001/QCL
CSA: Jean-Francois Lepage, Defence Scientist, 418-844-4000 Ext.: 4192

The scientific or technical validity of this Contract Report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Defence R&D Canada – Valcartier
Contract Report
DRDC Valcartier CR 2011-167
March 2011

Canada

Synthetic Infrared Scene

Improving the KARMA IRSG module and signature modelling tool SMAT

Marc-André Labrie
Eric Rouleau
Jonathan Richard
Mathieu Desmeules
Alexandre Bastien
Louis Tanguay Informatique inc.

Geoffroy Rivet-Sabourin
Technologie Intelligence Image inc.

Prepared By:

Louis Tanguay Informatique inc.
825 Boulevard Lebourgneuf, Bureau 204
Québec, Canada G2J 0B9

Contractor's Document Number: LTI-SIS-2011-1
Contract Project Manager: Marc-André Labrie
PWGSC Contract Number: W7701-082234/001/QCL
CSA: Jean-Francois Lepage, Defence Scientist, 418-844-4000 4192

The scientific or technical validity of this Contract Report is entirely the responsibility of the Contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Defence R&D Canada – Valcartier

Contract Report
DRDC Valcartier CR 2011-167
March 2011

Principal Author

Original signed by Marc-André Labrie

Marc-André Labrie

Project Manager

Approved by

Original signed by Jean-Francois Lepage

Jean-Francois Lepage

Contract Scientific Authority

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2011

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2011

Abstract

The main objective of the contract “Synthetic Infrared Scene” (W7701-082234) was to increase the level of fidelity of infrared guided weapon engagement simulations inside the KARMA simulation environment. The work was carried out from November 2008 to March 2011. This contract report focuses on presenting the new functionalities that were added to the infrared scene generator (IRSG) module which is part of the KARMA framework. Modifications were also done to the signature modelling and analysis tool (SMAT) which uses the IRSG to perform various kind of analysis.

Résumé

L'objectif principal du contrat "Scène Infrarouge Synthétique" (W7701-082234) a été d'augmenter le niveau de fidélité d’engagements impliquant des autodirecteurs infrarouges dans l'environnement de simulation KARMA. Le travail a été réalisé à partir de novembre 2008 jusqu’à mars 2011. Ce rapport de contrat est axé sur la présentation des nouvelles fonctionnalités qui ont été ajoutées au module de génération de scène infrarouge (IRSG) faisant partie de l’environnement KARMA. Des modifications ont également été apportées à l’outil de modélisation et d'analyse de signature infrarouge (SMAT) qui utilise l’IRSG pour effectuer différents types d'analyse.

This page intentionally left blank.

Executive summary

Synthetic Infrared Scene: Improving the KARMA IRSG module and signature modelling tool SMAT

M.-A. Labrie; E. Rouleau; J. Richard; M. Desmeules; A. Bastien; G. Rivet-Sabourin; DRDC Valcartier CR 2011-167; Defence R&D Canada – Valcartier; March 2011

The main objective of the contract "Synthetic Infrared Scene" (W7701-082234) was to increase the level of fidelity of infrared guided weapon engagement simulations inside the KARMA simulation environment. The work was carried out from November 2008 to March 2011. This contract report focuses on the new functionalities that were added to the infrared scene generator (IRSG) module which is part of the KARMA framework. Modifications were also done to the signature modelling and analysis tool (SMAT) which uses the IRSG to perform various kind of analysis.

The main improvements to the IRSG module include: the use of advanced rendering libraries and mechanisms to exploit graphical processor units, better atmospheric modelling including the use of a wideband correlated-k mode for increased performances, better representation of backgrounds, better representation of surface reflections, implementation of a zoom antialiasing algorithm, and representation of scattering effects. The SMAT tool was improved to take account of the new IRSG features and to add new modelling abilities.

These improvements should reflect in the ability to build better signature models and ultimately in an increased fidelity of the generated scene for a wider range of conditions (including atmospheric conditions, engagement geometry, etc.). This will contribute to a significant increase of fidelity of the results obtained within the KARMA framework.

The work done within this contract was aimed at fully digital simulation, so increasing performances was not driving the development. Eventually, some aspects, such as the zoom antialiasing algorithm, may need to be optimized to increase the frame rate.

Sommaire

Synthetic Infrared Scene: Improving the KARMA IRSG module and signature modelling tool SMAT

M.-A. Labrie; E. Rouleau; J. Richard; M. Desmeules; A. Bastien; G. Rivet-Sabourin; DRDC Valcartier CR 2011-167;R & D pour la défense Canada – Valcartier; Mars2011.

L'objectif principal du contrat "Scène Infrarouge Synthétique" (W7701-082234) a été d'augmenter le niveau de fidélité d'engagements impliquant des autodirecteurs infrarouges dans l'environnement de simulation KARMA. Le travail a été réalisé à partir de novembre 2008 jusqu'à mars 2011. Ce rapport de contrat est axé sur la présentation des nouvelles fonctionnalités qui ont été ajoutées au module de génération de scène infrarouge (IRSG) faisant partie de l'environnement KARMA. Des modifications ont également été apportées à l'outil de modélisation et d'analyse de signature infrarouge (SMAT) qui utilise l'IRSG pour effectuer différents types d'analyse.

Les principales améliorations apportées au module IRSG comprennent: l'utilisation de bibliothèques de rendu avancé et des mécanismes permettant d'exploiter les processeurs de rendu graphique, une meilleure modélisation de l'atmosphère incluant l'utilisation d'un mode à large bande basé sur les k-corrélés pour des performances accrues, une meilleure représentation des arrière-plans, une meilleure représentation des réflexions sur les surfaces, la mise en œuvre d'un algorithme d'anticrénelage (*zoom antialiasing*), et la représentation des effets de diffusion. L'outil SMAT a été amélioré afin de tenir compte des nouvelles caractéristiques de l'IRSG et d'ajouter de nouvelles capacités de modélisation.

Ces améliorations devraient se refléter dans la capacité de construire des signatures de modèles de meilleure qualité et, en fin de compte, en une fidélité accrue de la scène générée pour un plus large éventail de conditions (y compris les conditions atmosphériques, la géométrie de l'engagement, etc.) Cela contribuera à une augmentation significative de la fidélité des résultats obtenus avec KARMA.

Le travail effectué durant ce contrat visait à améliorer les simulations entièrement numériques : l'aspect consistant à améliorer les performances n'était pas ce qui a dirigé les efforts. Éventuellement, certains aspects pourraient être optimisés pour augmenter la fréquence des images générés, tels que l'algorithme d'anticrénelage développé.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

The main objective of the contract "Synthetic Infrared Scene" (W7701-082234) was to increase the level of fidelity of infrared scenes to be used in infrared guided weapon engagement simulations within the KARMA simulation environment. The work was carried out from November 2008 to March 2011. This contract report focuses on the new functionalities that were added to the infrared scene generator (IRSG) module which is part of the KARMA framework. Complementary information about the recent modifications is also detailed in [1]. More details about the state of the IRSG prior to this contract can be found in [2]. The report also presents modifications to the signature modelling and analysis tool (SMAT). This tool is used to build the signature models by populating the model databases, and generating various kind of analysis through images generated by the IRSG module. The current version of SMAT is 3.11. The first iteration of development, which produced the version 2.00 of SMAT prior to this contract, is presented in details in [3] and [4].

The major modifications to the IRSG and SMAT are divided as described below. The architectural review of the IRSG is presented in Section 2. The efforts done to improve the low-level rendering techniques are documented in Section 3. The computation of apparent radiance and the addition of reflections caused by the sun and the background on scene's models are discussed in Section 4. An atmospheric module based on MODTRAN was also integrated in the KARMA framework in order to improve the atmospheric parameters; its mechanisms are detailed in Section 5. The antialiasing technique developed to improve the results of the IRSG is presented in Section 6. Important modifications were also done related to the background of generated images to produce non-uniform textures, as presented in Section 7. New parameters were also included in the temperature and material database of a model used in the IRSG process (Section 8). Degradation of images due to atmospheric scattering was also implemented, such as described in Section 9. Other minor modifications which are helpful in data generation, analysis, etc. are also presented to keep track of the changes made during this contract (Sections 10, 11, 12 and 13).

2 Scene generation refactoring

Scene generation is done either through a KARMA simulation or through the SMAT modelling tool. At the beginning of this contract, a KARMA service (`KARMA::SceneGenerator3D`) was used by both "clients" and was based on other simulation services such as `KARMA::Theatre`, `KARMA::Environment`, etc. This approach was quite straightforward for a simulation as the scene was driven by models and parameters of a scenario. However, SMAT was required to emulate a simulation by creating models and parameters similarly to a KARMA scenario. This task was done in SMAT by the `SMAT::AnalysisGenerator` class which invokes the scene generation module. Therefore, SMAT was tightly coupled to the simulation framework. To remove unwanted dependencies, the architecture of the scene generation module had to be revisited.

2.1 IRSG

The scene generation functionalities have been isolated into the `KARMA::IRSG` class and an application programming interface (API) has been created to allow controlling the scene and the rendering parameters. Such an interfacing requires to create a lot of methods and data storage to foster the usability and flexibility of the IRSG.

The IRSG still performs rendering of the infrared (IR) scene using OpenSceneGraph (OSG) for the 3D scene management. It is almost a standalone application as it is independent from the KARMA simulation, but the IRSG is available as a part of the `KARMA::SceneGenerator3D` library. The following KARMA libraries are required: `KARMA::Materials`, `KARMA::DataTypes`, `KARMA::AdvancedTypes` and `KARMA::Coordinates`. The IRSG would be further isolated from the KARMA simulation framework by using its own library.

2.2 Adapting for KARMA

The scene generation service in KARMA has been easily adapted to the IRSG. Indeed, most of the source code of the `KARMA::SceneGenerator3D` has been relocated into the IRSG. The responsibility of the `KARMA::SceneGenerator3D` is now to adapt the IRSG for a KARMA simulation instead of implementing scene generation, as shown in Figure 1.

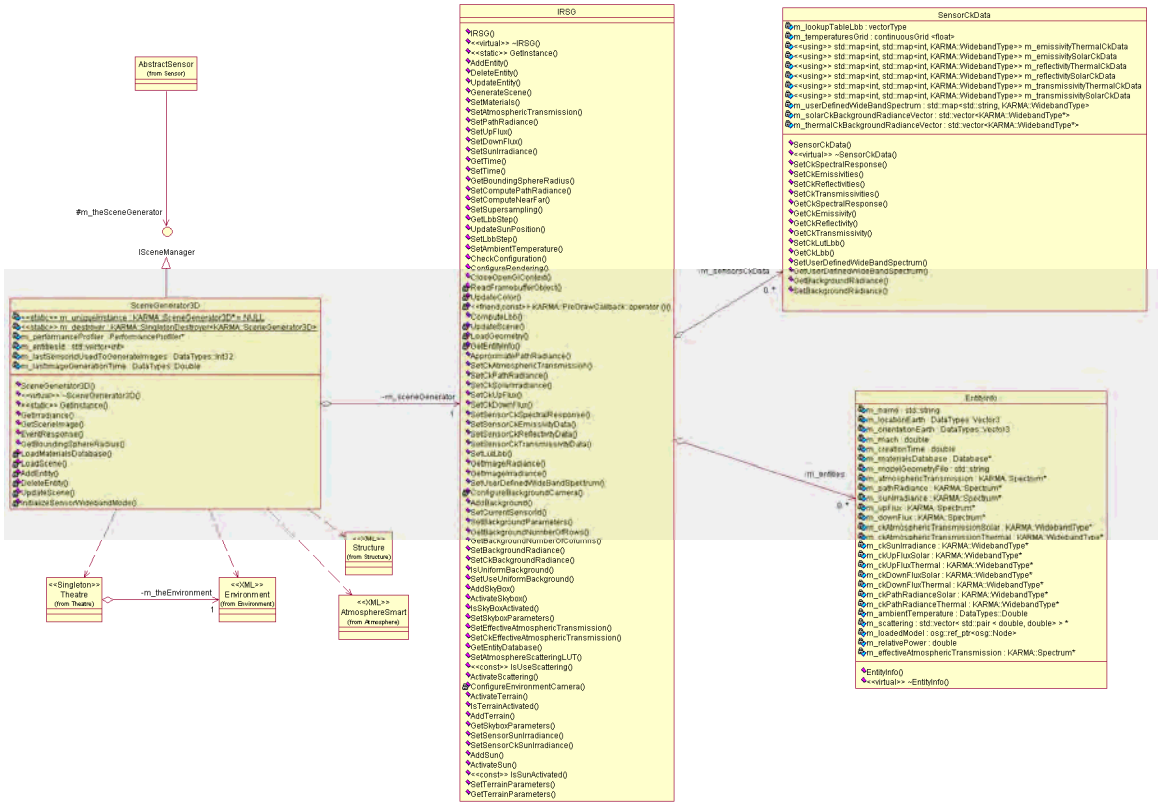

Figure 1: Architecture of the IRSG adapted for KARMA.

2.3 Adapting for SMAT

In SMAT, scene generation is still invoked by the `SMAT::AnalysisGenerator` class, but now using the `KARMA::IRSG` class instead of the `KARMA::SceneGenerator3D` class. Similarly to the `KARMA::SceneGenerator3D`, the `SMAT::AnalysisGenerator` adapts the IRSG for infrared analysis in SMAT. The use of the IRSG allowed to reduce the dependencies on KARMA. Besides the libraries required by the IRSG, the `SMAT::AnalysisGenerator` uses some KARMA libraries to gather appropriate parameters for the rendering (e.g. `KARMA::SmartAdapter`, `Scattering`), as shown in Figure 2. The method `SMAT::AnalysisGenerator::SetKarmaParameters()` is used to gather these parameters and configure the IRSG accordingly.

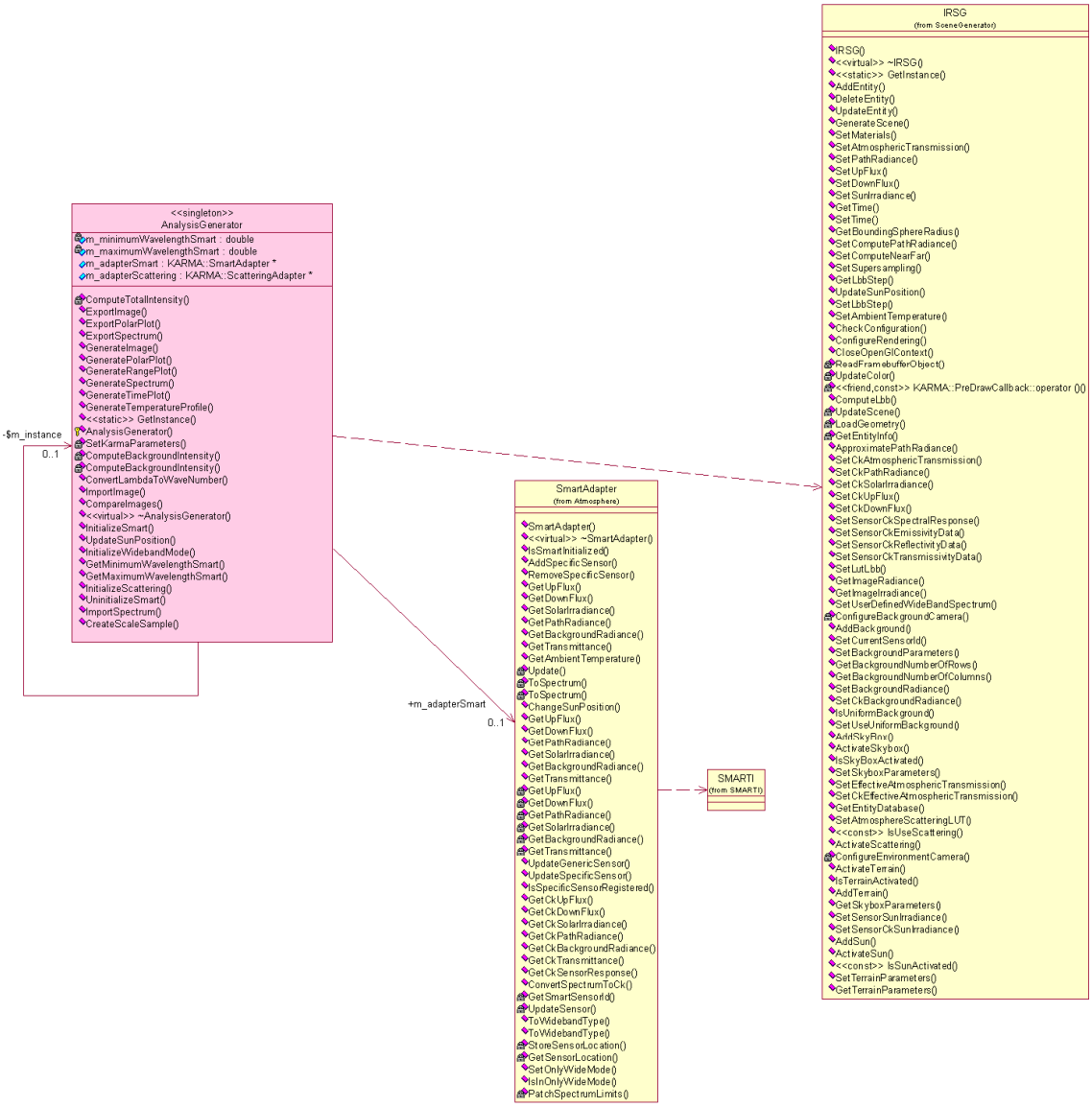

Figure 2: Architecture of the IRSG adapted for SMAT.

2.3.1 GUI uncoupling

Basically, the purpose of the `SMAT::AnalysisGenerator` is to operate and configure the IRSG to perform infrared analysis. In order to increase reusability of this class and allow automated testing, the `SMAT::AnalysisGenerator` has been revisited to remove any dependencies to the graphical user interface (GUI) of SMAT. The following classes have been gathered into a library named `SmatCore`:

- the `SMAT::Settings` class acts as a container for the SMAT application settings, including the scene generation settings;

- the `SMAT::AnalysisData` class acts as a container for the input parameters and the output results of an infrared analysis; and

- the `SMAT::AnalysisGenerator` class acts as a manager of the scene generation settings and performs various infrared analyses based on images generated by the IRSG: image, intensity spectrum, polar plot, time plot and range plot.

When a SMAT analysis is created, the `SMAT::AnalysisGeneratorDialog` class is the GUI that gathers the input parameters of the analysis, stores the corresponding information in a `SMAT::AnalysisData` object and triggers the `SMAT::AnalysisGenerator` to start an infrared analysis.

3 Advanced rendering techniques

Infrared scene generation has to deal constantly with rendering issues. It is desirable that the scene generation performance and scene realism increase in order to obtain accurate data while the simulation length is not too much impacted. Naturally, the rendering libraries and mechanisms used to accomplish the scene generation process have a crucial impact. The modifications done to the IRSG module in regards to low-level rendering are described in this section.

3.1 Migrating from OSG 2.2.x to OSG 2.8.0

OSG is an open source 3D graphical toolkit that is used to manage and render 3D models in visible and infrared modes. The toolkit is written in C++ and is based on the lower level API of Open Graphic Library (OpenGL). In order to be compatible with Delta3D 3.2, which is used by the KARMA 3D viewer, and to benefit of the enhancements (including several bug fixes) of the latest version at this time, the OSG library used by the IRSG was updated to version 2.8.0.

The most notable change of this migration was the replacement of the `osgUtil::SceneView` class, which has been declared deprecated, by the `osgViewer::Viewer` class. Table 1 summarizes the updated methods corresponding to the deprecated methods.

Table 1: Mapping between deprecated and updated methods of OSG 2.8.0.

Deprecated methods (`osgUtil::SceneView`)	**Updated methods**
`setDefault`	None
`setClearColor`	`osg::Camera::setClearColor`
`setSceneData`	`osgViewer::setSceneData`
`cull`	`osg::Viewer::renderingTraversals`
`setComputeNearFarMode`	`osg::Camera::setComputeNearFarMode`
`getCamera`	`osgViewer::getCamera`
`update`	`osgViewer::eventTraversal` `osgViewer::updateTraversal`
`setFrameStamp`	`osgViewer::setFrameStamp`
`setViewport`	`osg::Camera::setViewport`
`setViewMatrix`	`osg::Camera::getViewMatrix`
`draw`	`osgViewer::renderingTraversals`
`flushAllDeletedGLObjects`	None

Since the `osgViewer::renderingTraversals()` method performs both the cull and the draw operations, the call to the `IRSG::UpdateColor()`method has been placed in a pre-draw callback. This allows updating the color after the culling and before the draw.

3.2 Migrating from Mesa to OpenGL

OpenGL is an API for performing 3D rendering. However, it is not a software product and it does not have any source code. OpenGL is only a specification that describes an interface and its expected behaviour. Therefore, to use the OpenGL API, an OpenGL implementation is required.

On Windows operating system (OS), a very basic implementation of OpenGL (1.1 or 1.4 depending of the OS version) is provided through `opengl32.dll`. It is important to understand that this standard Windows library alone does not provide any hardware acceleration. In order to get hardware acceleration and implementation of the newer OpenGL specification (1.5 to 4.1), video drivers from graphics card manufacturer (like AMD/ATI, Intel, NVIDIA) need to be installed. These drivers are called OpenGL Installable Client Driver (ICD). Installing a video driver will not replace `opengl32.dll`: it is a system file and belongs to Windows (meaning that only Microsoft may update it). When a video driver is installed, another file will be copied on the system (`nvoglv32.dll` in the case of NVIDIA) and the registry will be modified. Then, `opengl32.dll` will call into the real GL driver (`nvoglv32.dll`). The OpenGL runtime accesses the registry to determine which ICD to load.

For programmers, installing video card drivers will not make `gl.h` or `opengl32.lib` available. Those are files that come with the compiler and there are no updated `gl.h` and `opengl32.lib` files. These are stuck at GL 1.1 and are likely to be forever. This means that it is not possible to link directly to any function provided by newest OpenGL versions or extensions. In order to use these newer functions, `glext.h` and `wglext.h` shall be used to get function pointer at runtime with the `wglGetProcAddress` call. Fortunately, some libraries such as GLEW and GLEE were developed to make available function pointers.

Mesa 3D is an open-source implementation of an API which is very similar to OpenGL. In fact, the Mesa 3D implementation tries to follow the OpenGL specification (with a certain delay in comparison to graphic card manufacturers) but does not guarantee the respect of it. Table 2 presents pros and cons of using Mesa 3D.

Table 2: Pros and cons of using Mesa 3D.

Pros	Cons
- Completely platform independent since rendering is done via software. - The offscreen rendering API (OSMesa), in 16/32 bits, is easy to use.	- OSMesa API is software only. - Need to use its own `OpenGL32.dll` (not the standard Windows one). - Does not support all OpenGL specifications. - Not up-to-date with the latest OpenGL release. - Small developer community.

Initially, the KARMA high dynamic range rendering was implemented using the Mesa's off-screen rendering API (OSMesa). The OSMesa interface supports 16-bit and 32-bit color channels rendering into user-allocated blocks of memory.

An important requirement of this project was to migrate from Mesa 3D to OpenGL ICD in order to gain access to hardware acceleration and latest features of OpenGL. Table 3 presents the pros and cons of using OpenGL ICD.

Table 3: Pros and cons of using OpenGL ICD.

Pros	Cons
- Hardware accelerated rendering. - Up-to-date with the OpenGL features. - Use the standard Windows `Opengl32.dll`. - Up-to-date with the GLSL features. - Large community of users. - Used by OSG.	- Availability of certain features (OpenGL extension) is hardware dependant. - Computation results may be slightly different depending on hardware.

A good OpenGL implementation will render with hardware acceleration whenever possible. However, the implementation is free to render without hardware acceleration. OpenGL does not provide a mechanism to ensure that an application is using hardware acceleration, nor to query that it is using hardware acceleration.

3.3 Framebuffer object

Usually, images have a 8-bit format for each channel (red, green and blue). Thus, each component value can range from 0 to 255. In some situations, like in the case of radiance calculation for infrared scene generation, this range is not sufficient i.e. does not provide enough possible values. Framebuffer objects (FBO) allow storing images with a high-dynamic range (HDR), in 32-bit floating point values. Without HDR, clipping will occurs, meaning that areas that are too dark will be totally black (RGB(0,0,0)) and areas that are too bright will be totally white (RGB(1,1,1)).

Thus, the OpenGL FBO extension has been used to perform the off-screen rendering into a 32-bit floating-point texture. By default, OpenGL uses the default framebuffer as the final rendering destination. Everything that is put in this buffer is automatically drawn to the screen. On the other hand, the FBO mechanism allows generating an image (2D array of pixels) in a buffer (other than the default OpenGL framebuffer) which can be post-processed. The image is not necessarily drawn on the screen: only if it is instructed to do so. Table 4 shows how to initialise the floating point texture within OSG.

Table 4: Definition of an off-screen floating-point texture in OSG.

```
int imageWidth = 500;
int imageHeight = 500;
osg::TextureRectangle* offscreenTexture = new osg::TextureRectangle;
offscreenTexture->setTextureSize(imageWidth, imageHeight);
offscreenTexture->setSourceFormat(GL_RGBA);
offscreenTexture->setSourceType(GL_FLOAT);
offscreenTexture->setInternalFormat(GL_RGBA32F_ARB);
```

Table 5 shows how to configure an `osgViewer::Viewer` to render into a floating point texture instead of using the standard framebuffer.

Table 5: Render to texture using FBO in OSG.

```
// Get the main camera from the viewer
osg::Camera* camera = viewer->getCamera();

// Set the camera to render in the FBO
camera->setRenderTargetImplementation(osg::Camera::FRAME_BUFFER_OBJECT);

// Attach a texture to the FBO
camera->attach(osg::Camera::COLOR_BUFFER, offscreenTexture.get(), 0, 0, false,
0, 0);
```

3.3.1 Context conflict

Unlike in OSMesa, a FBO needs an OpenGL context to get working. A conflict between the SMAT/IRSG OpenGL contexts appeared when a new OpenGL context was created for the FBO. The solution retained to avoid this conflict was to memorize and make current the right context when calling the scene generation of the IRSG from SMAT. Table 6 shows the code used to avoid the OpenGL context conflict.

Table 6: Code to avoid conflicts between the SMAT/IRSG OpenGL contexts.

```
HGLRC KARMA::IRSG::m_glContext;
HDC KARMA::IRSG::m_deviceContext;

// Memorize the current OpenGL context
m_deviceContext = m_openglWindowContext->getHDC();
m_glContext = m_openglWindowContext->getWGLContext();

…

// Activate the OpenGL context of the IRSG (avoids conflicts with SMAT)
if(wglGetCurrentDC() != m_deviceContext ||
   wglGetCurrentContext() != m_glContext)
{
    wglMakeCurrent(m_deviceContext, m_glContext)
}
```

3.4 KARMA architecture

Figure 3 and the following text show the modifications that were done to the KARMA architecture to make possible the changes discussed previously in this section of the report.

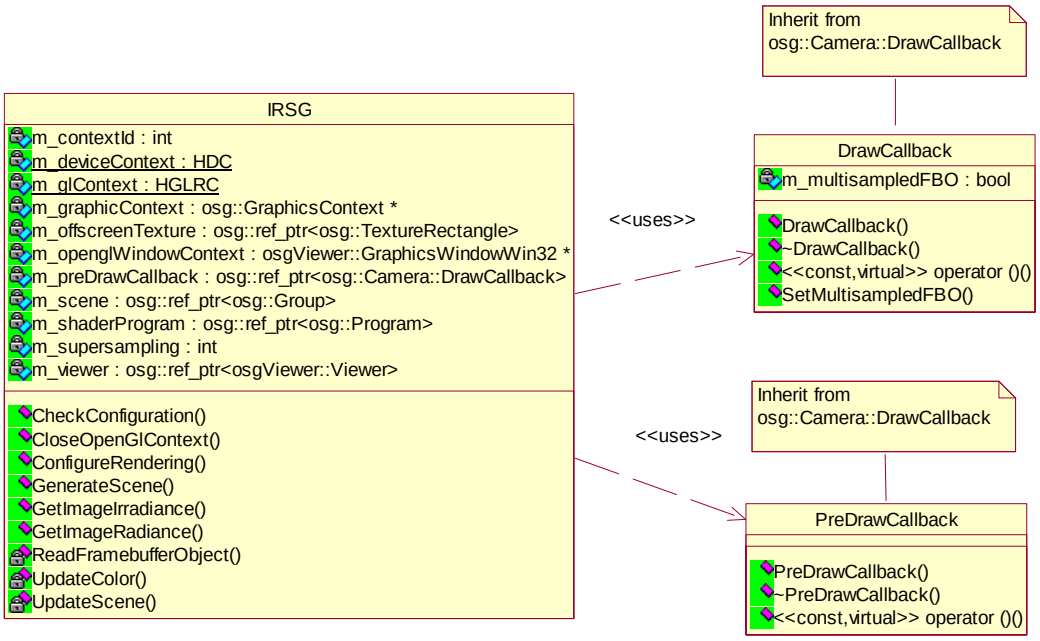

Figure 3: Architecture of the IRSG related to the HDR hardware rendering.

- `IRSG` is the class which handle the scene generation based on the `osgViewer::Viewer`. It implements the high dynamic range hardware rendering using an off-screen texture, a FBO (via the `osgViewer::Viewer` camera settings) and a shader program. It also takes care of the OpenGL context manipulation.

- The `operator()` method of the `PreDrawCallback` is executed after the culling operation and before the draw operation. It has the responsibility of calling the `IRSG::UpdateColor()` method.

- The `operator()` method of `DrawCallback` is executed after the draw operation. Its responsibility is to bind the right FBO to be read by the `IRSG::ReadFrameBufferObject()` method.

4 Apparent radiance and reflections

The addition of reflections to the IRSG module's computations is important to bring realism to infrared (IR) scenes. As mentioned in [5], various components (like the sun, the background, flares and plumes) can produce reflections on other entities in a scene, which can have an impact on an IR sensor.

During this contract, the computation of apparent radiance has been improved to vary spatially and to include contributions from reflections of the sun and fluxes (from the sky and the ground). Figure 4 shows how these reflections are combined to the emitted radiance of a spherical 3D model. These contributions are calculated and modulated for each rendered pixel, based on pixel's normal compared with the sun position (vector oriented toward the sun); and the zenith vector for fluxes.

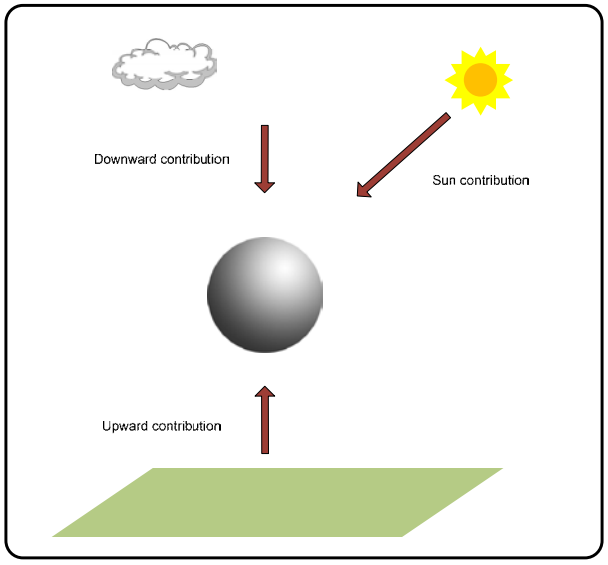

Figure 4: Different reflections added on a 3D model.

Since the contributions of the apparent radiance depend on the geometry, the *shader* technology is used. It was a natural choice because it is possible to access the normal (interpolated from the surrounding vertices' normal) of each rendered pixel of the 3D model in order to combine and modulate these contributions at the rendering stage. Obviously, doing computations for each pixel can add an overhead, but this process is done via the graphics processing unit (GPU) which accelerates the overall process. Indeed, modern GPUs are very efficient to make operations in parallel such as those described previously.

The vertex shader presented in Table 7 is used to compute the normal and position of each vertex composing the 3D model. These values are interpolated in the fragment shader to obtain the normal and position for each fragment located between vertices.

Table 7: Vertex shader used in the IRSG.

```
varying vec3 normal;
varying vec3 position;

void main(void)
{
    normal = normalize(gl_NormalMatrix * gl_Normal).xyz;
    position = normalize(gl_ModelViewMatrix * gl_Vertex).xyz;
    gl_Position = ftransform();
}
```

The in-band apparent radiance of thermal, path and reflected components are calculated in the IRSG while the fragment shader presented in Table 8 is used to compute the final value of pixels of a rendered 3D model, taking into account the polygon-geometry dependant components. This value represents the combination of the three components. These components, initially computed such as described in [5], are now computed as described below.

The thermally emitted contribution to the apparent radiance as seen through the atmosphere is now given by

$$L_{app}^{therm} = \varepsilon_{scaling} \cos^{N}(\theta) \int \varepsilon_{surf}(\lambda) L_{bb}(T_{surf}, \lambda) \tau_{path}(z, \lambda) R_{sens}(\lambda) \, \mathrm{d}\lambda , \tag{1}$$

where $\varepsilon_{\mathrm{scaling}}$ is an emissivity scaling that is detailed in Section 12 and N is the *N angle factor* detailed in Section 8.3.

The apparent radiance component from the atmosphere located between the surface and the sensor is still given by

$$L_{app}^{path} = \int L_{path}(\lambda) R_{sens}(\lambda) \, \mathrm{d}\lambda . \tag{2}$$

The apparent radiance component from reflections on a surface is now given by

$$\begin{aligned} L_{app}^{refl} = & \frac{1}{\pi}\left(\frac{\theta_{i\,zenith}}{\pi}\right) \int \psi^{+}(\lambda) \rho_{surf}(\lambda) \tau_{path}(\lambda, z) R_{sens}(\lambda) \, \mathrm{d}\lambda \\ & + \frac{1}{\pi}\left(1 - \frac{\theta_{i\,zenith}}{\pi}\right) \int \psi^{-}(\lambda) \rho_{surf}(\lambda) \tau_{path}(\lambda, z) R_{sens}(\lambda) \, \mathrm{d}\lambda , \\ & + \frac{1}{\pi} \cos(\theta_{i\,sun}) \int E_{sun}^{0}(\lambda) \rho_{surf}(\lambda) \tau_{path}(\lambda, z) R_{sens}(\lambda) \, \mathrm{d}\lambda \end{aligned} \tag{3}$$

where $\psi^{+}(\lambda)$ is the upward flux, $\psi^{-}(\lambda)$ is the downward flux and $E^{0}_{sun}(\lambda)$ is the sun irradiance.

The `uniform` parameters at the beginning of the shader are values which come from the IRSG while the `varying` parameters are the result of a previous computational stage done in the vertex shader.

Table 8: Fragment shader used in the IRSG.

```
// different contributions used to calculate reflected
// radiance at target surface position
uniform float LUpRefApp;
uniform float LDownRefApp;
uniform float LSunRefApp;

// sun position in eye coordinates
uniform vec3 sunPositionEye;

// zenith unit vector in eye coordinates
uniform vec3 zenithVecEye;

// thermal radiance
uniform float LThermApp;

// path radiance
uniform float LAtmApp;

// transparency
uniform float transparency;

// factor affecting LThermAPP
uniform float nFactor;

////////////////////////////////////////////////////////////
// Inputs from pixel shader (interpolated)
////////////////////////////////////////////////////////////

varying vec3 normal;
varying vec3 position;

void main(void)
{
     ////////////////////////////////////////////////////////////
     // Constants
     ////////////////////////////////////////////////////////////

     float PI = radians(180.0);
     float PI_ON_TWO = radians(90.0);

     vec3 fragmentNormal = normalize(normal); //Important: after interpolation normal modulus != 1.
     vec3 fragmentPosition = normalize(position);

     ////////////////////////////////////////////////////////////
     // Thermal radiance
     ////////////////////////////////////////////////////////////

     float viewFactor = 0.0;
     vec3 viewVec = normalize( -position );

     // Ensure that the dot product is between the interval [-1.0, 1.0]
     float dotProduct = max(-1.0, min(1.0, dot(viewVec, fragmentNormal)));

     float viewAngle = acos(dotProduct);

     // Compute view factor when view angle is not 90 degrees from the normal (0 otherwise)
     if(viewAngle != PI_ON_TWO)
     {
         viewFactor = 1.0;

         if(nFactor != 0.0)
         {
             // Compute view factor using cosine of angle between view vector and surface's normal
             viewFactor = pow( abs(dotProduct), nFactor );
         }
     }

     // Modulate thermal radiance using view factor
     float LThermAppContribution = viewFactor * LThermApp;

     ////////////////////////////////////////////////////////////
     // Sun contribution
```

```
    ////////////////////////////////////////////////////////////
    float LSunRefAppContribution = 0.0;

    vec3 sunVec = normalize(sunPositionEye-position);

    // Ensure that the dot product is between the interval [-1.0, 1.0]
    dotProduct = max(-1.0, min(1.0, dot(sunVec, fragmentNormal)));

    float sunAngle = acos(dotProduct);

    // Solar reflection is visible only when the view vector is on the surface's side where the reflection occurs
    if(sign(cos(viewAngle)) == sign(dotProduct))
    {
        // Modulate sun contribution using cosine of angle between sun and surface normal
        LSunRefAppContribution = abs(dotProduct) * LSunRefApp;
    }

    ////////////////////////////////////////////////////////////
    // Upward and Downward contributions
    ////////////////////////////////////////////////////////////

    // Ensure that the dot product is between the interval [-1.0, 1.0]
    dotProduct = max(-1.0, min(1.0, dot(zenithVecEye,fragmentNormal)));

    float angleZenith = acos(dotProduct);

    // Contributions are inverted when the view vector is on the surface's backside
    if(viewAngle > PI_ON_TWO)
    {
        angleZenith = PI - angleZenith;
    }

    // Modulate contribution using angle between zenith and surface normal
    float LUpRefAppContribution = (angleZenith/PI) * LUpRefApp;

    // Modulate contribution based on angle between zenith and surface normal
    float LDownRefAppContribution = (1-(angleZenith/PI)) * LDownRefApp;

    ////////////////////////////////////////////////////////////
    // Reflected radiance
    ////////////////////////////////////////////////////////////

    float LRefAppContribution = LUpRefAppContribution + LDownRefAppContribution + LSunRefAppContribution;

    ////////////////////////////////////////////////////////////
    // Total apparent radiance = Ltherm + Lref + Latm
    ////////////////////////////////////////////////////////////

    // total perceived radiance
    float computedColor = LThermAppContribution + LRefAppContribution + LAtmApp;

    // set the fragment color
    gl_FragColor = vec4(computedColor, computedColor, computedColor, 1.0-transparency);
}
```

In order to supports double-sided surfaces, the fragment shader uses the view angle θ_{view}, defined as the angle between the view vector and the surface's normal, when it modulates `LThermApp`, `LSunRefApp`, `LUpRefApp` and `LDownRefApp` components:

- Thermal radiance `LTherm` is the same whether the surface is seen from front ($\theta_{view} < 90°$) or behind ($\theta_{view} > 90°$).

- Sun reflection `LSunRefApp` is not seen (component set to 0) when the surface is seen from the opposite side where the reflection occurs.

- Upward `LUpRefApp` and downward `LDownRefApp` reflections contributions are interchanged when the surface is seen from behind ($\theta_{view} > 90°$).

Notice that the vertex and fragment shaders described in Table 7 and Table 8 are directly embedded in the code, in the `IRSG.cpp` file (i.e. they are not defined in external files). Figure 5

shows the effect of reflections on a platform: a nice gradient effect is visible. The sun is in front of the platform with an elevation of 45.0 degrees (above horizon).

Figure 5: An example showing images obtained without (left) and with (right) reflections.

4.1 Reflections in KARMA simulations

The reflections are always taken into account (i.e. activated) in KARMA simulations. The sun azimuth and elevation are defined in the `Environment` model and can be modified in its XML parameters file, as shown in Table 9.

Table 9: KARMA's environment parameters related to the sun.

```
<parameter name="SunAzimuth">
    <double>0</double>
    <documentation>
         The sun azimuth in radians measured East from
         North. This parameter is used by the visual environment (display
         the sun in a viewer) and can also be used by the simulation. The
         value range is from 0 to 2PI.
    </documentation>
</parameter>
<parameter name="SunElevation">
    <double>0.7854</double>
    <documentation>
         The sun elevation in radians. This parameter is
         used by the visual environment (display the sun in a viewer) and
         can also be used by the simulation.  The value range is from -PI/2
         to +PI/2.
    </documentation>
</parameter>
```

4.2 Reflections in SMAT

SMAT does not have access to the `Environment` model. However, the sun parameters can be modified via the *Analysis Settings* dialog box as show in Figure 6.

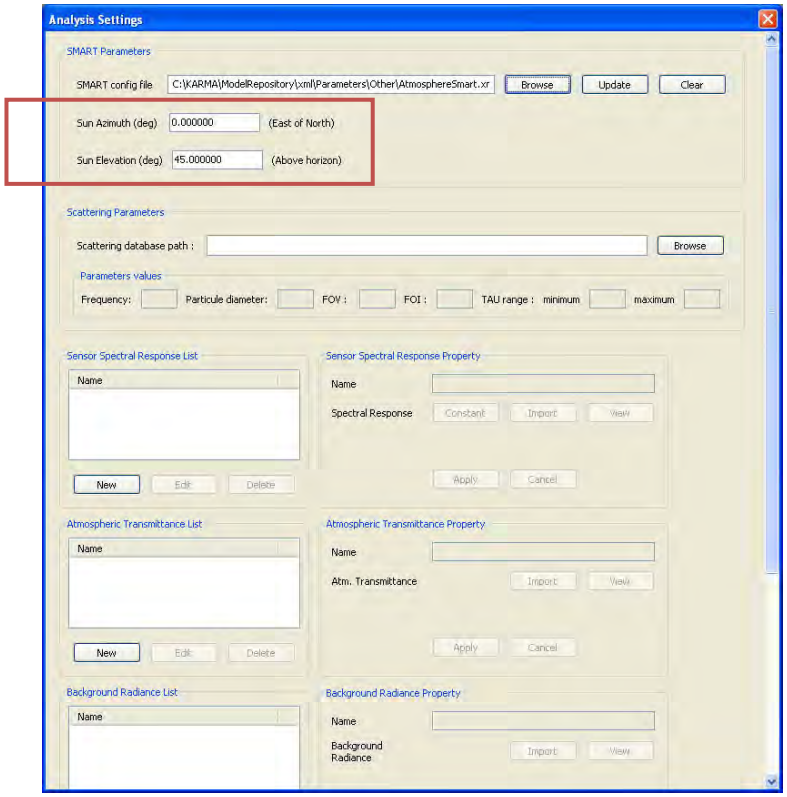

Figure 6: Defining the sun parameters inside SMAT.

Figure 7 shows the parameters of an analysis, sun and fluxes, causing the reflections. They can be activated or deactivated, based on the values selected. On the figure, the value selected (i.e. SMART which is described in Section 5), refers to the atmospheric model used to compute the components. It can be noted that, for backward compatibility and work using simple atmosphere models, a geometry independent reflection such as described in [5] can be computed by selecting *Approx.* in those fields.

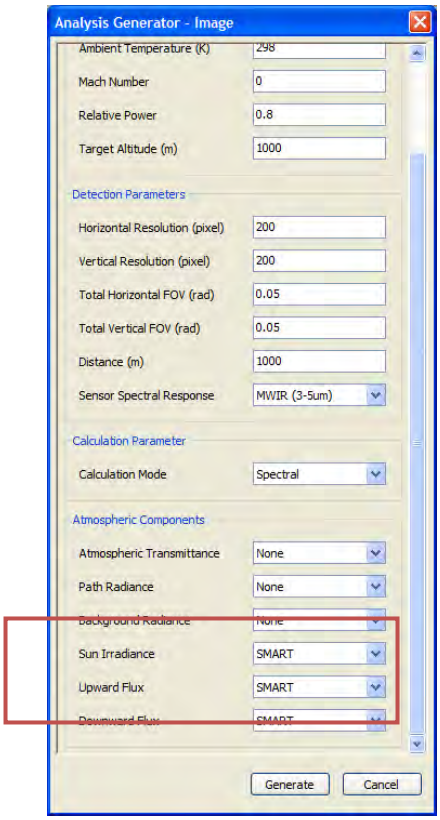

Figure 7: Parameters causing reflections in SMAT.

5 Atmospheric module

To improve IR scenes computed in the IRSG, an objective of this project was to use MODTRAN to obtain precise radiometric values depending of the current environment conditions to be recreated in the scene. There was already an effort at DRDC-Valcartier to produce a C++ library which is built on top of MODTRAN. This library is named SMART (not to be confused with SMAT) for *Suite for Multi-resolution Atmospheric Radiative Transmission* [6][7]. An interface, SMARTI, is also available to facilitate communications with another software. The main purpose of SMART is the calculation of atmospheric values such as transmitted solar irradiance, atmospheric fluxes, path and background radiances, and transmittance. This library can produce outputs in both spectral or wideband correlated-k (CK) format.

5.1 Environment refactoring

The `Environment` is a model already defined in KARMA. What needed to be added is the concept of atmospheric model. Thus, to integrate this new concept, the `Environment` model was modified to increase the modularity: the `Environment` gathers `Atmosphere` objects which are able to calculate atmospheric parameters depending on the current context (see Figure 8).

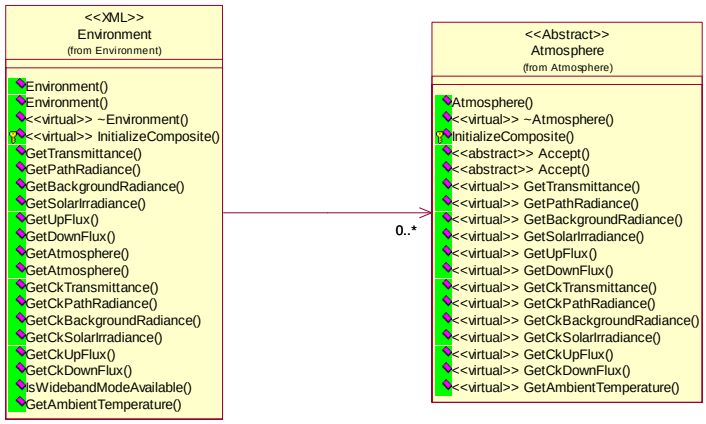

Figure 8: The relation between Environment and Atmosphere.

At this moment, the values which can be calculated by an `Atmosphere` are:

- transmittance;
- path radiance;
- sky (or background) radiance;
- solar irradiance;
- upward flux;
- downward flux; and
- ambient temperature.

`Environment` traverses its list of atmospheric models and returns the appropriate data corresponding to the first atmospheric model found that meet the selection criteria. For each data mentioned previously which is required, `Environment` verifies, via the `Accept()` method, that the atmospheric model can provide this kind of data and uses an appropriate spectral band for processing this data. For example, the following model (Table 10) can handle six data types (with their correlated-k version):

Table 10: An example of Accept() method for an atmospheric model.

```
DataTypes::Boolean KARMA::AtmosphereExample::Accept(DataTypes::DoublewavelengthMin,
                                                    DataTypes::DoublewavelengthMax,
                                                    int property)
{
    if (property != KARMA::Atmosphere::Transmittance_Type&&
        property != KARMA::Atmosphere::CkTransmittance_Type&&
        property != KARMA::Atmosphere::PathRadiance_Type&&
        property != KARMA::Atmosphere::CkPathRadiance_Type&&
        property != KARMA::Atmosphere::BackgroundRadiance_Type&&
        property != KARMA::Atmosphere::CkBackgroundRadiance_Type&&
        property != KARMA::Atmosphere::SolarIrradiance_Type&&
        property != KARMA::Atmosphere::CkSolarIrradiance_Type&&
        property != KARMA::Atmosphere::UpFlux_Type&&
        property != KARMA::Atmosphere::CkUpFlux_Type&&
        property != KARMA::Atmosphere::DownFlux_Type&&
        property != KARMA::Atmosphere::CkDownFlux_Type)
             return false;

    // See if the range is ok
    if (m_baseWavelengthMin <= wavelengthMin && m_baseWavelengthMax >= wavelengthMax)
         return true;
    else
         return false;
}
```

5.2 An atmospheric model based on SMART

To make available the SMART library into KARMA based simulations, an atmospheric model, `AtmosphereSmart`, was defined and is presented in Figure 9.

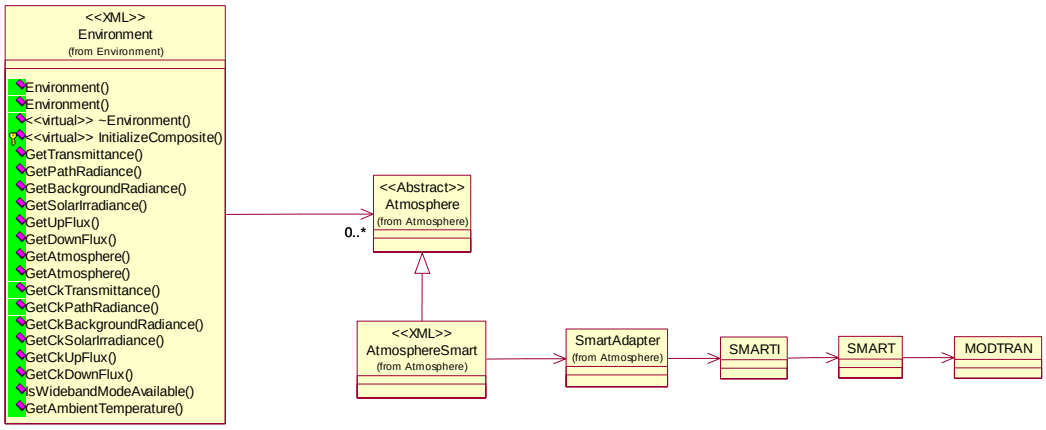

Figure 9: Model class diagram for the KARMA AtmosphereSmart atmospheric model.

- `AtmosphereSmart` is the atmospheric model developed based on SMART which can be used in KARMA simulations to obtain values from SMART. A XML file contains the parameters which characterize this model (see file example shown in Annex A).

- `SmartAdapter` is used to call methods from SMART and transform data from the IRSG to a SMART compliant format (and vice versa). An object of this class manages the complexity of the manipulation of objects and data (e.g. coordinate system conversion) inside SMART. Thus, `AtmosphereSmart` is not aware of how SMART works. Notice that SMAT uses this component directly (i.e. SMAT does not use the `AtmosphereSmart` model) when SMART values are required.

- `SMARTI` is the interface to the `SMART` library which is based on MODTRAN. Notice that SMART is using a precise version of MODTRAN: MODTRAN 4 version 3 release 1, available in `%KARMA_ROOT%\Softwares\Modtran`.

The `KARMA::Spectrum` data type is used to manipulate the SMART spectral format. For the wideband-ck mode, a wrapper was created (`WidebandType`) which encapsulates SMART wideband types (`NormTypeCk`, `RadTypeCk`, `NormType`).

For KARMA simulations, the `AtmosphereSmart` model shall be defined in the composition of an `Environment` model, as shown in Table 11; its parameter file will allow initializing the atmospheric model in an appropriate way.

Table 11: Adding an atmospheric module in the Environment's composition.

```
<composite name="SMARTI">
     <component>AtmosphereSmart</component>
     <parameters>$(KARMA_ROOT)\ModelRepository\xml\Parameters\
                 Other\AtmosphereSmart.xml
     </parameters>
     <composition>none</composition>
     <priority>0</priority>
     <documentation>
          Atmospheric model based on SMART/MODTRAN allowing the calculation of
          transmittance, path radiance, background radiance, solar irradiance, up
          flux, down flux; in spectral and/or wideband correlated-k.
     </documentation>
</composite>
```

Inside SMAT, even though `SmartAdapter` is used directly without accessing the `AtmosphereSmart` model, the atmospheric settings are still described by the parameters file of `AtmosphereSmart`. Then, the parameters file of this model must be configured properly and set as shown in Figure 10 in order to initialize SMART appropriately.

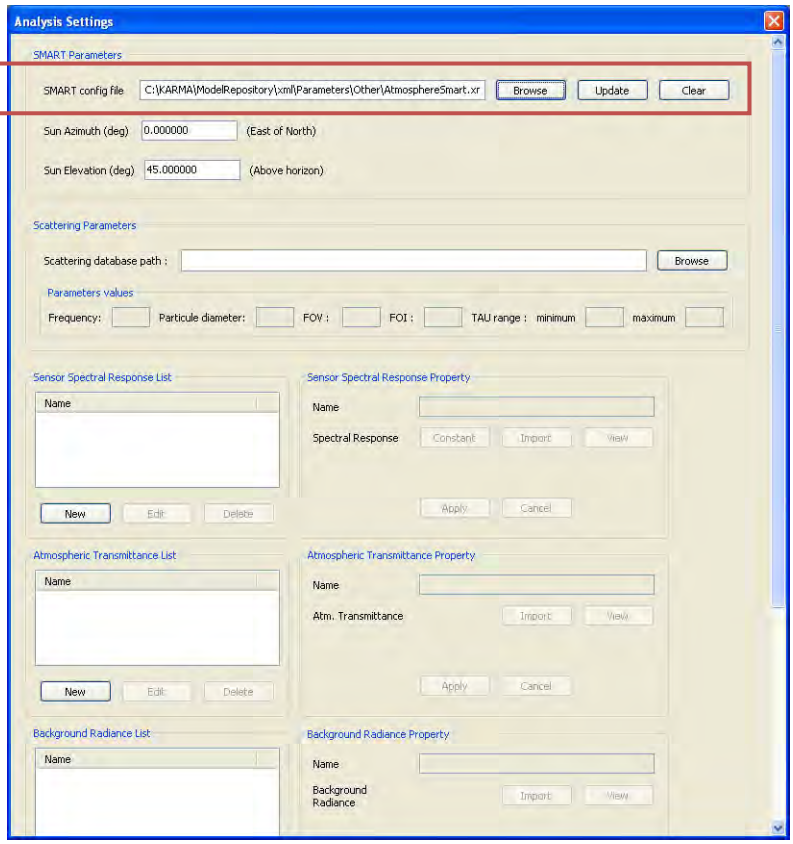

Figure 10: Setting the SMART configuration file inside SMAT.

5.2.1 Initialisation review

For the wideband-ck mode, a pre-calculation phase must take place. Before beginning the execution of a simulation, each spectrum necessary for the scene generation process are converted into their CK equivalent. Thus, for each sensor, the following data are converted using the sensor's conversion space:

- the spectral response of the sensor (solar and thermal);
- spectral characteristics (reflectivity, transmissivity, emissivity) of materials (solar and thermal);
- lookup table of blackbody radiance as a function of temperature (wideband-ck): LBB (T, λ) →Lbb_therm (T, ck). The table is constructed, using a SMART data structure, with a minimum value of 200 K and a maximum value of 2200 K with steps of 12.5 K. SMART returns an interpolated value if the table is interrogated with a temperature in between two table values.

A container class, `SensorCkData`, is used to store the converted data and is available to the IRSG module.

5.2.2 Calculation mode selection

Inside KARMA simulations, the calculation mode (spectral or wideband-ck) is defined for each `ImagingSensor` and set inside its parameters file. If the parameter `CalculationMode` is set to `1`, the sensor uses the spectral mode; while the value `2` indicates that the sensor uses the wideband-ck mode. When the wideband-ck mode is selected, the apparent radiance components are computed in wideband format using the SMART data structure, the wideband atmospheric quantities returned by SMART, and the previously converted properties, as presented in Section 5.2.1.

Inside SMAT, a combo box inside an analysis dialog box (Figure 11) allows the user to set if the process shall be executed in wideband-ck or in spectral mode.

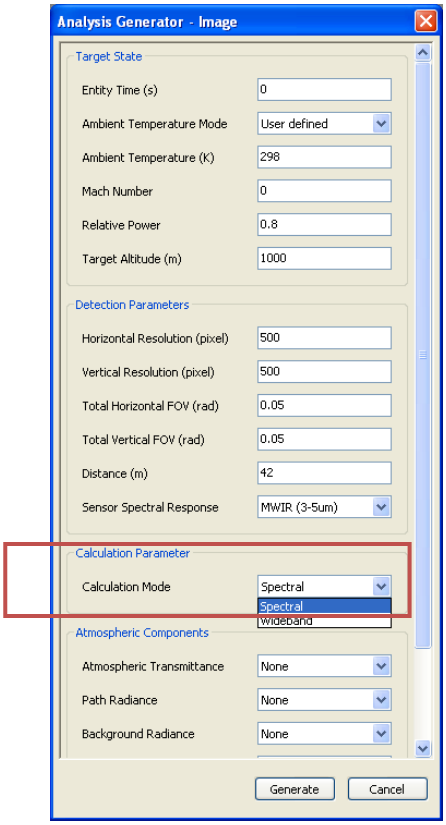

Figure 11: Setting the execution mode (wideband-ck/spectral) in SMAT during an analysis.

6 Antialiasing

One limit of infrared scene generation is that platforms can be located very far in front of infrared sensors and subtend only few pixels or less than one pixel. Depending on the sampling method used, platforms may have a poor representation, or even worst, not be represented at all on the screen, even if the sensor is supposed to detect energy (referred as the apparent radiance). This will lead to radiometric inaccuracies, affecting sensors based on the IRSG and therefore, the results of the engagements.

Sampling is the process during which a continuous function is mapped on a discrete one. This mechanism is used in computer rendering process to map analog data on a digital system. Since a pixel is the smallest unit of a computer graphic (it is filled or not i.e. cannot be partly filled), it will then produce jagged edges on objects where the pixel grid is visible. Sampling methods can also produce *scintillation* phenomenon when an object is moving and crossing pixel boundaries, producing a rapid variation of the color.

To reduce these phenomena, antialiasing techniques were developed. Among these, a technique more appropriate for the purpose of infrared scene generation is called *zoom antialiasing* (ZAA).

6.1 Zoom antialiasing technique

Rather than supersampling over the entire screen, the zooming window technique allows supersampling areas of the screen that are susceptible of causing aliasing artefacts. Ref. [8] describes a procedure that renders only portions of the screen at higher resolution. More recently, [9] describes the implementation of the ZAA procedure in the GPU, which is named CgAA. The implementation is based on the Cg shader programming language and uses multiple rendering passes for reducing visual artefacts caused by aliasing.

ZAA is much more efficient than full screen antialiasing (FSAA) because it focuses on areas of the screen where aliasing is more likely to occur. However, the implementation is more complicated since the algorithms are not embedded into the hardware (i.e. graphics card). Nevertheless, ZAA gives more control on the antialiasing technique and more precision regarding the calculated apparent radiance.

In brief, ZAA consists to 1) produce high definition images (i.e. textures with way more samples than the number of pixels covered by the model) of each scene model; 2) produce accurate low definition images (i.e. textures with a number of texels near the number of pixels covered by the model) for each high definition image; and 3) substitute scene elements by their corresponding low definition images. This technique is described with more details in the following sub-sections.

6.1.1 Method description

A basic approach based on the models' bounding sphere was developed and may eventually be upgraded to obtain more accurate results. This technique is inspired by the approach presented in [9]. Notice that GLSL was chosen over CG because it is natively supported by OSG.

Phase 1: Produce high definition images of each scene's element.

General approach

For each scene's entity, i.e. 3D model, a high definition image is produced.

Technical considerations

For each 3D model in the scene, a zoom camera (`osg::Camera`) is created. The area covered by the bounding sphere in the main camera's view is calculated in term of pixels. The zoom camera will adjust its field of view based on this value: if the value is a power of two, this value is taken; otherwise, the field of view is increased to reach a dimension (always a square) corresponding to a power of two. Notice that the minimum FOV is 2x2 pixels. A more accurate way would have been to zoom on the exact area covered by the model (see Figure 12) but this technique is more complicated to implement. The downsampling process used in the following phase is eased given the fact that the image generated by the zoom camera is square and a power of two.

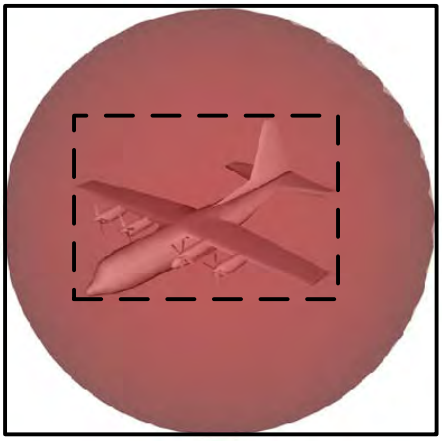

Figure 12: Zoom camera view based on bounding sphere (solid line) compared to view based on exact model extends (dashed).

After the zoom camera is set, it renders its view into a texture. Notice that there is no background (e.g. skybox, sun) in the zoom camera view: it is only viewing one 3D model. This also means that a zoom camera does not see two models in its view if models are one beside the other or one behind the other. The size of this texture depends on the user selection. The current system allows selecting a dimension of 512x512, 256x256 or 128x128 pixels, depending on the performance and precision desired. The mechanism that allows choosing the size is explained in Section 6.1.2. Notice that the origin of the zoom camera(s) is the same as the main camera to respect the object perspective (see Figure 13).

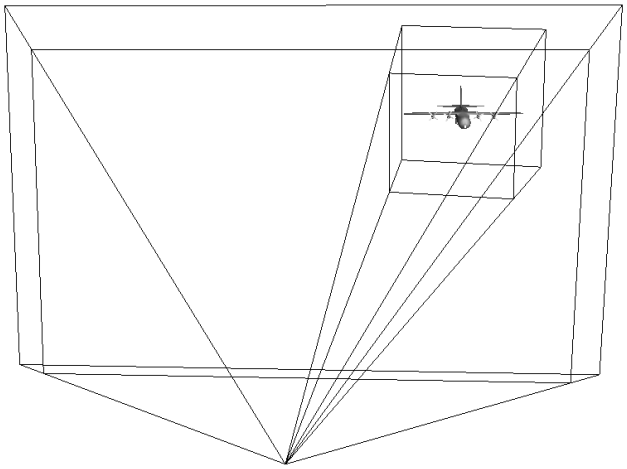

Figure 13: An example showing frustra for the main scene's camera and a zoom camera.

Phase 2: Post-process high definition images.

General approach

Each high definition image (texture) is reduced to the size the image must fit in.

Technical considerations

Shaders are used during this phase to regroup texels in order to reduce images to the size mentioned in the previous phase. Two shaders were defined for the downsampling process. The first one allows using 4 samples of the input texture (2x factor); and the second one, by using 16 samples of the input texture (4x factor) as demonstrated in Figure 14.

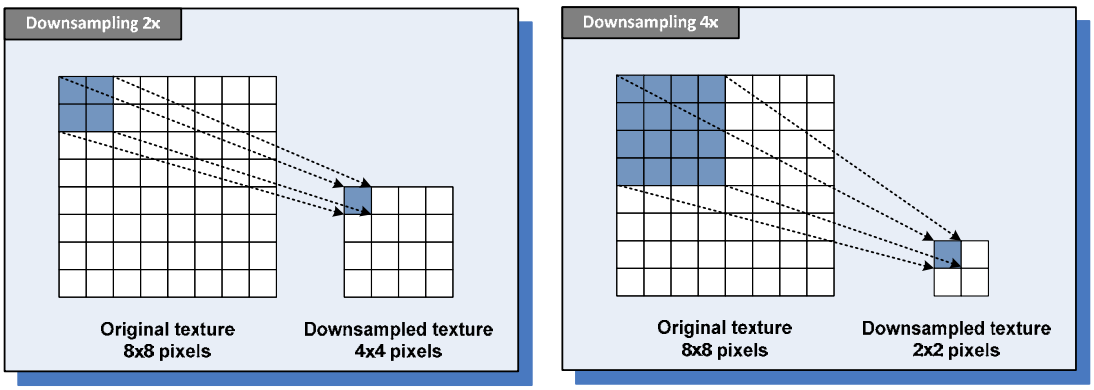

Figure 14: General downsampling process (2x and 4x).

Thus, the texel value (i.e. color) in the downsampled texture is the average of 4 or 16 samples, depending on the downsampling factor. By applying multiple combinations of these shaders, one after another, it is possible to reduce the input texture (which is a power of two) to the size of the

output texture (which is also a power of two) as shown in Figure 15. Consequently, a maximum of 4 passes are necessary to downsample the worst case scenario: from 512 pixels to 2 pixels (512 $\rightarrow_{/4}$ 128 $\rightarrow_{/4}$ 32 $\rightarrow_{/4}$ 8 $\rightarrow_{/4}$ 2).

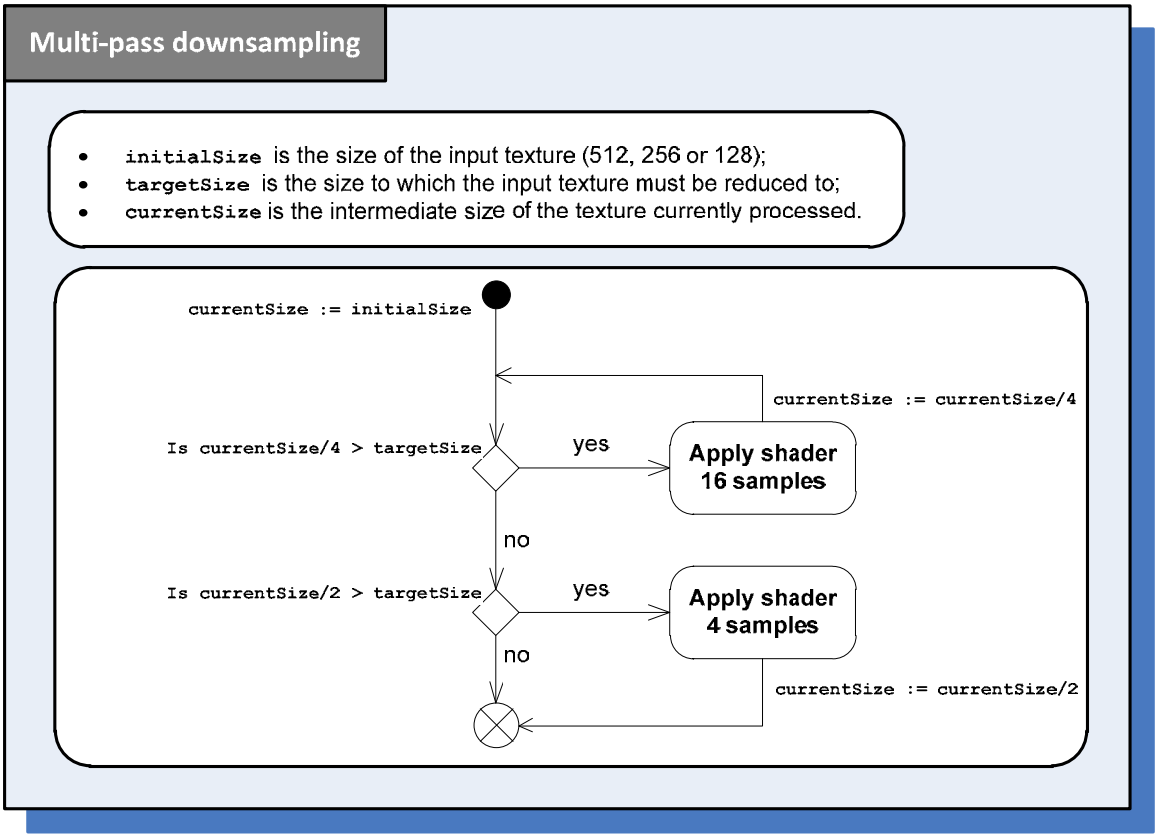

Figure 15: Multi-pass downsampling with shaders.

The downsampling task is realized via OSG objects with the help of shaders and cameras. An example of the process is depicted in Figure 16. This example shows how to downsample from a 512x512 pixels texture to a size of 128x128 pixels.

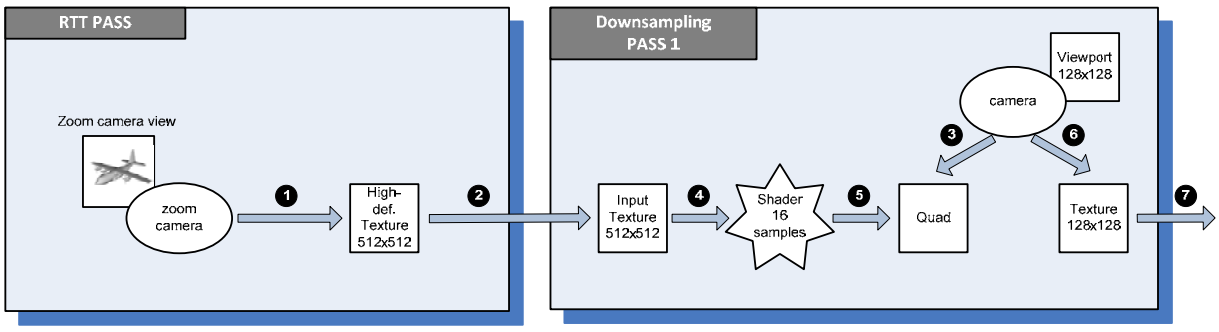

Figure 16: Downsampling process via OSG.

The steps introduced in the previous image are explained below.

1. A zoom camera takes a snapshot of its associated 3D model in high definition: 512x512 pixels (could also be 256x256 or 128x128 depending on the user selection).

2. This high definition texture is the input of a first downsampling pass.

3. A new camera is created and its viewport is set to the desired downsampling size i.e. 128x128 (as mentioned previously, ZAA's shaders can reduce the input size by a factor 2x or 4x). A quad is added as a child to this camera to receive the downsampled texture.

4. The high definition texture is an input of the shader.

5. The shader downsamples the texture and assigns a color to each fragment of the quad.

6. The camera takes a snapshot of the quad, producing the downsampled texture.

7. This new texture is the final one in this example, but it could have been the input of another downsampling pass if required.

Phase 3: Replace each scene's element by its low definition image.

General approach

For each element of the scene, create a quad, which shall replace the original 3D model, facing the camera; and apply the low definition texture generated previously on that quad.

Technical considerations

The ZAA capacity uses `osg::Billboard` in order to create a quad in the scene which will hold a texture to be displayed. Basically, billboards are quads which rotate around an axis or a point. They are often used in 3D environments to represent a tree, rotating about its Z axis to give the impression of a 3D object to an observer located on the ground. For the ZAA needs, quads will rotate about their center to always face the camera. The advantage for the ZAA capacity is that it does not need to calculate rotation to be applied on quads for each frame, since this is automatically done via the internal processing of billboards.

To ease the process of replacing a model by its corresponding quad, the system takes advantage of the level-of-detail (LOD) mechanism. In general, a LOD object uses a model with a higher number of polygons when located near the camera and switches to a model containing fewer polygons when moving away from the camera (range based). Another kind of LOD, least known, is also available, which is more appropriate to the context of antialiasing. This second LOD bases its "switch" mechanism (from a model to another one) on the size of the initial model bounding sphere diameter into a camera's view. The result is the same as the range based LOD: passed a certain distance, the LOD will display a different geometry. However, it is easier and more

intuitive to specify that when a model covers X pixels, to activate the ZAA and replace it by a quad. Notice that the minimum quad size displayed is 2x2 pixels.

In the current implementation, three modes can be selected by the user:

1. ZAA 128 pixels;

2. ZAA 256 pixels; and

3. ZAA 512 pixels.

The mode selected specifies the dimension of the camera's viewport but also the size at which the LOD will replace the 3D model by a quad (threshold value as depicted in Figure 17). For example, if the first mode is selected i.e. ZAA 128 pixels, the "high definition" image generated by the zoom camera will be 128x128 pixels and the 3D model will be replaced by a quad when the model's bounding sphere dimension is less than 128 pixels.

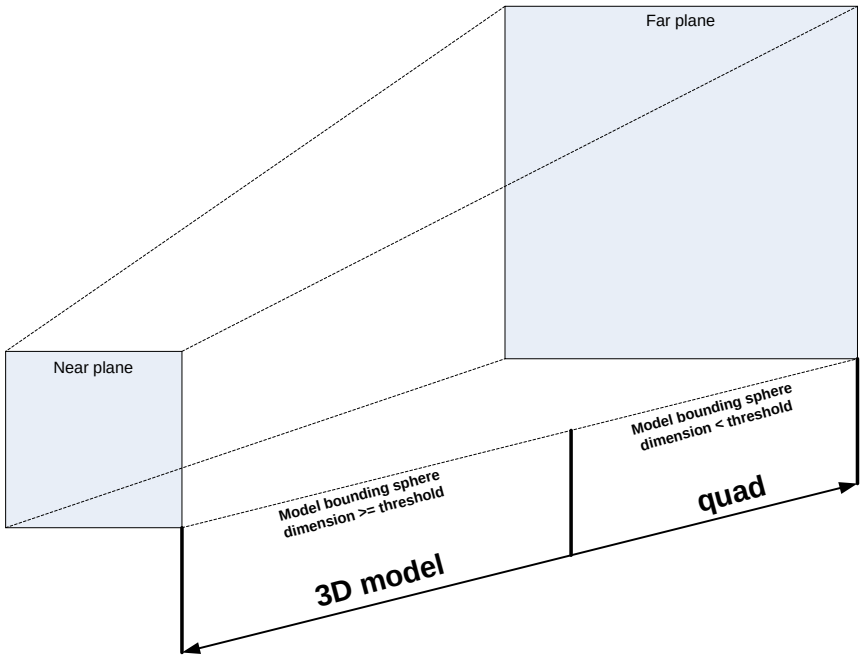

Figure 17: Switching between the 3D model and a quad during ZAA process.

Finally, a render to texture phase will produce the final frame corresponding to the sensor's view. During the rendering process, each quad is rasterized, and fragments receive their color from the downsampled texture. Since there is not a 1 on 1 correspondence for texels of the generated texture and screen pixels (i.e. texels are the same size but do not line up exactly with the pixels of the display), there will be a sampling error when rendering the final frame. Various sampling methods are available with OpenGL (nearest, bi-linear, etc.). The sampling error is, of course, affected by the method selected and the choice of a sampling method compared to another will affect the performance of the rendering process. As a compromise between speed and precision, the bi-linear interpolation method was selected. Notice that a blending mechanism is used to

merge each quad's texture and the background in the generated image. The global ZAA process is resumed in Figure 18.

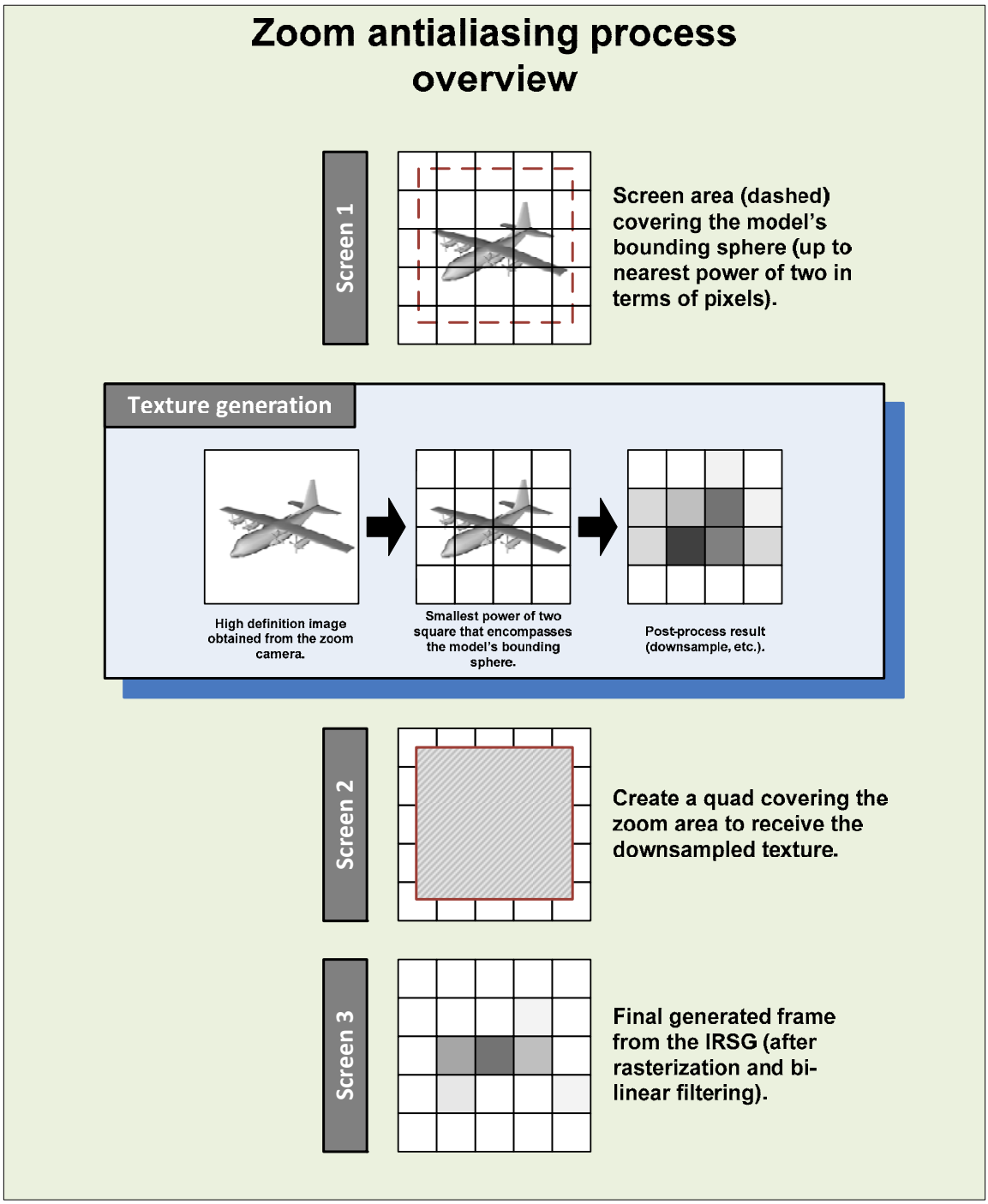

Figure 18: An overview of the ZAA process.

6.1.2 Zoom antialiasing activation

Inside the KARMA framework, it is possible to configure the antialiasing method for an `ImagingSensor` through its XML parameters file, using the parameter `AntialiasingMode` (contains a predefined value):

- `0`: None;
- `1`: Supersampling 2x;
- `2`: Supersampling 4x;
- `3`: Zoom antialiasing 128 pixels;
- `4`: Zoom antialiasing 256 pixels; and
- `5`: Zoom antialiasing 512 pixels.

Inside the SMAT tool, the *Options* dialog box shall be used to select the antialiasing mode which will be applied during the analysis as shown in Figure 19.

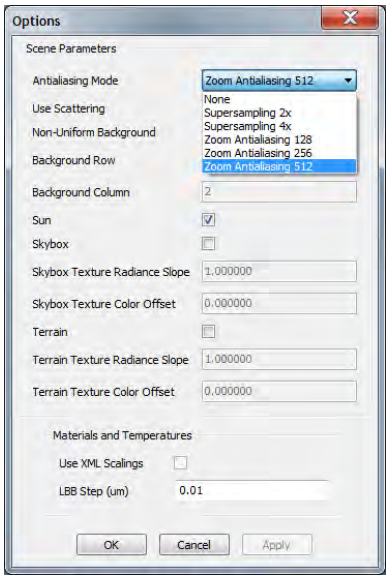

Figure 19: Zoom antialiasing activation within SMAT.

6.1.3 Troubleshooting

3D models may contains range based LOD(s). To be used with ZAA, it is highly recommended to remove any LOD(s) inside models with an appropriate model editor. Otherwise, the textures generated could be empty (except for the background) i.e. the model is hidden because it is located farther than the range defined by the maximal range LOD. Also, it is undesirable that the zoom camera renders an intermediate LOD i.e. a model containing fewer polygons than the full

details LOD. Eventually, LODs could be deactivated directly in the code or the range limit could also be increased.

6.1.4 Results

In this section, some results obtained via the SMAT tool are presented to show the performance and precision of the ZAA modes compared to other antialiasing mechanisms.

6.1.4.1 Performance

The time required to generate an image was monitored for each antialiasing methods; and when no antialiasing method (baseline) is applied. The CC130 model was used during those tests. Figure 20 shows the parameters used during the analysis. The distance at which the model is located from the sensor (`Distance` parameter in Figure 20) is changed to obtain results at various ranges. Rendering times are presented in Figure 21

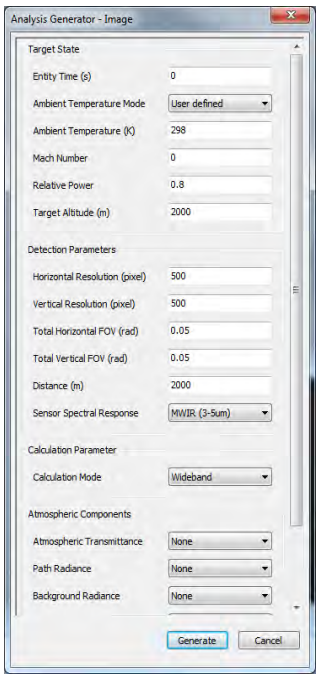

Figure 20: Parameters used for the ZAA performance analysis.

Figure 21: Time (ms) required to produce one image for the available antialiasing algorithms and when the platform (CC130) is located at various ranges (m).

6.1.4.2 Accuracy

To analyze the accuracy of antialiasing methods, the SMAT range plot analysis was used to generate graphs showing the contrast intensity as a function of range. The plots are generated without atmospheric attenuation and path radiance. The parameters used during those tests are shown in Figure 24. As a result, the variations of intensity with range represent the radiometric errors due to aliasing effect. The values obtained with the different antialiasing modes are presented in Figure 22. Considering that the results for the ZAA modes are not clearly visible, Figure 23 shows the results only for these modes. Notice that a simple sphere model, which has a radius of 1 meter, was used during those tests. It must be noted that the results are valid for the sensor settings (resolution and FOV) presented in Figure 24.

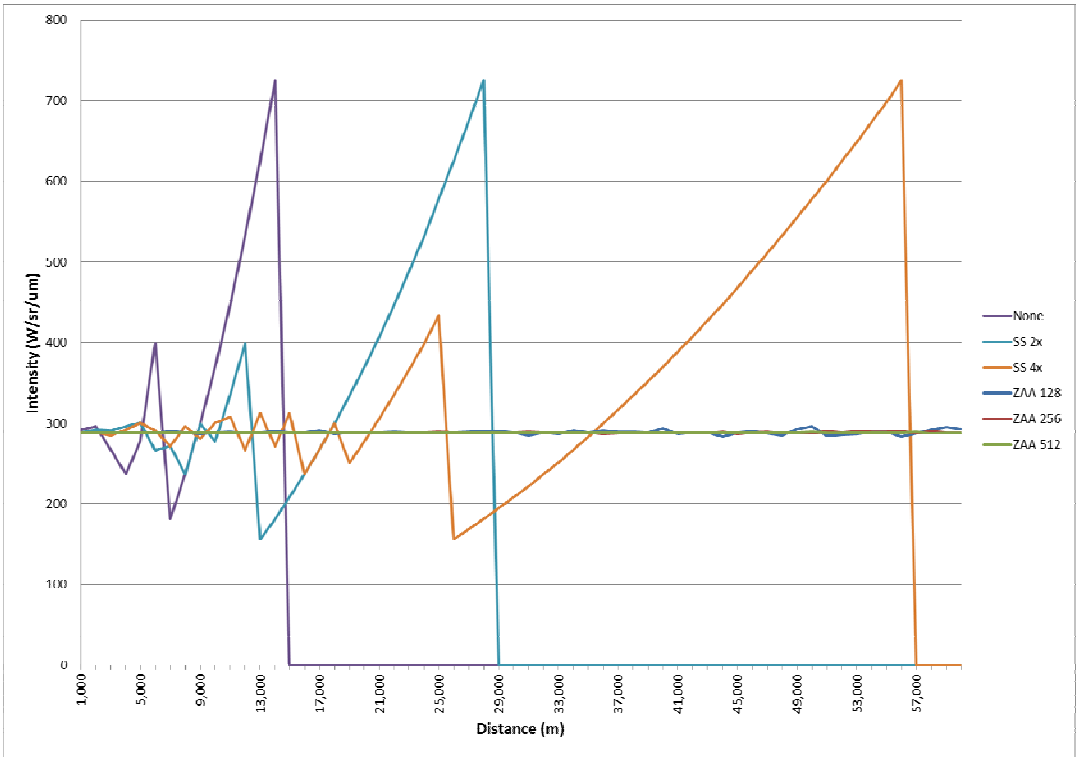

Figure 22: Contrast intensity vs. range for various antialiasing modes.

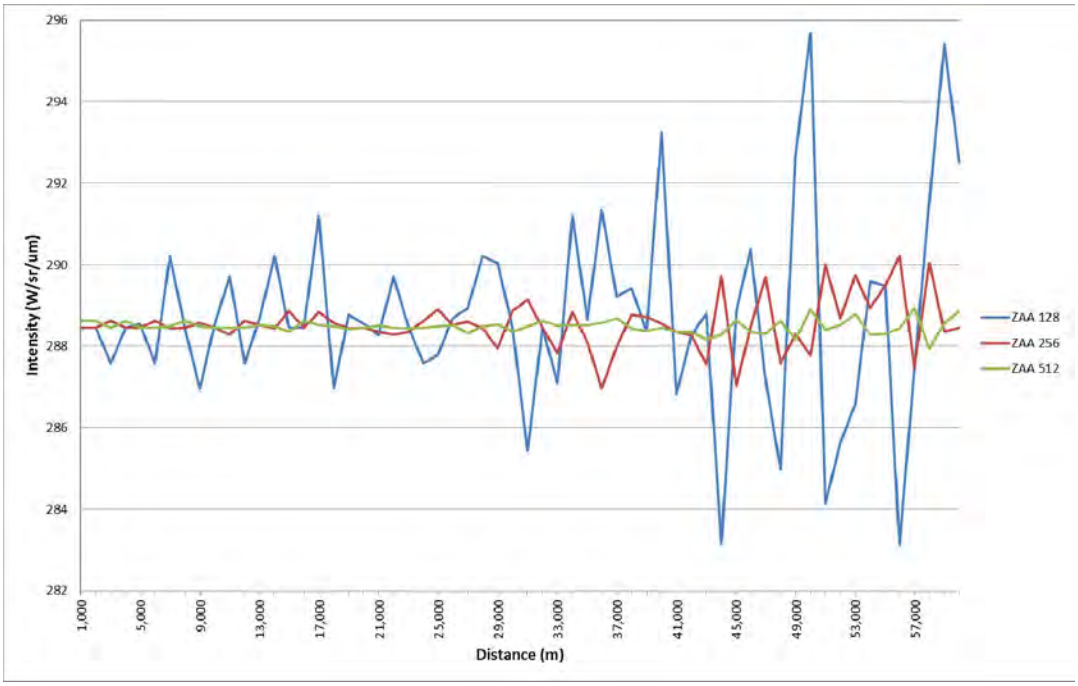

Figure 23: Contrast intensity vs. range for ZAA modes only.

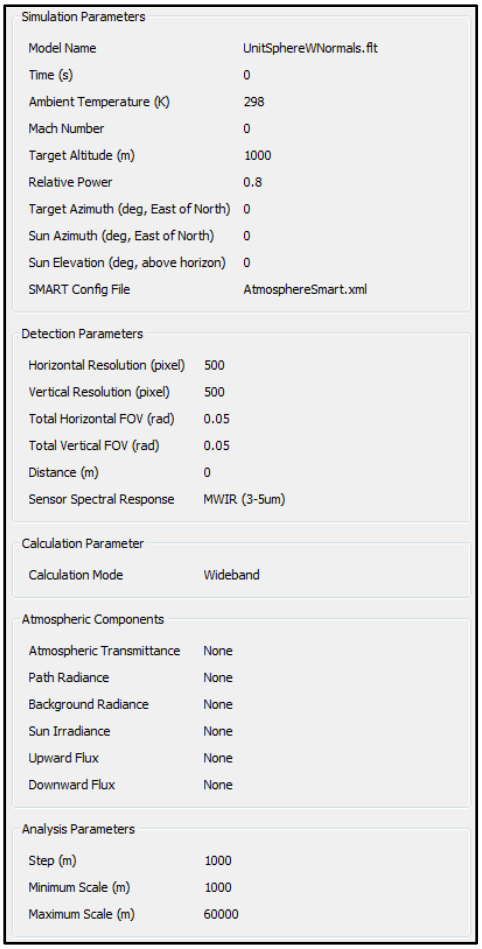

Figure 24: Parameters used for the ZAA accuracy analysis.

6.1.4.3 Discussion

The time required to produce an image, for the ZAA methods, increase when the platform is located far from the camera (c.f. Figure 21). Given the current implementation, this is expected because more downsampling passes are required as a model covers fewer pixels. At far distances, the ZAA methods take a lot of time to execute when compared to `Supersampling 4x`. In the context of infrared guided weapon engagement simulations (distance smaller than 10,000 m.), the rendering times for the ZAA methods are comparable to the supersampling mechanism. The first iteration of the ZAA development focused on mechanisms development to increase accuracy i.e. no performance optimizations were done. A second iteration would allow reducing execution times for the ZAA methods.

The contrast intensity values obtained at various ranges clearly indicates that the results obtained under a reasonable distance (60,000 meters) are better with the ZAA modes than the supersampling modes. Note that the 3D model covers 34.64 pixels in the sensor's view at 1000 meters, and 0.5774 pixel at 60,000 meters.

6.2 OSG rendering library

To implement the ZAA approach described in Section 6.1, new methods were required to manipulate OSG objects as well as some debugging methods. A library was developed to gather the new functionalities.

To ease reutilization and maximize modularity, the library is divided into three packages as depicted in Figure 25. A first package contains high-level functionalities for the ZAA capacity (`ZoomAntialiasing`). This package uses a second one containing methods allowing multi-pass rendering (`MultipassRendering`). These two packages use a third one which contains basic OSG methods to efficiently manage OSG objects (`OsgUtilities`).

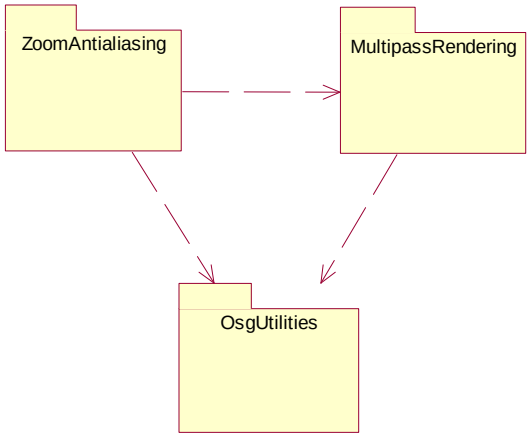

Figure 25: The packages defined in the OsgRendering library.

Figure 26 presents an overview of the classes defined in the library. These classes are explained in the following sections.

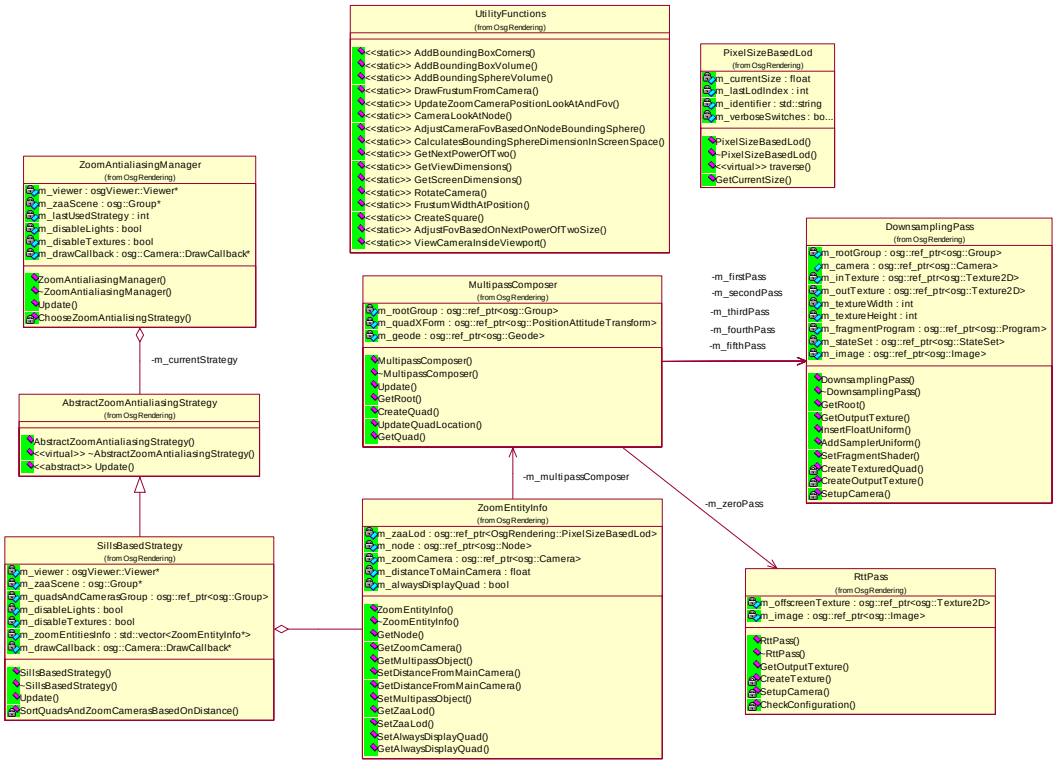

Figure 26: OsgRendering class diagram.

6.2.1 Zoom antialiasing capacity

As mentioned previously, a simple technique was developed but more sophisticated algorithms may also be deployed in the future. Thus, a system supporting multiple techniques is required. The algorithm may also have to be changed during a simulation to adapt to the current context. For example, it shall be possible to use an algorithm if an object subtends less than 1 pixel and another one if more than 1 pixel. To this end, a design pattern, named *Strategy pattern*, is used to ease the development by providing a mechanism to switch from one algorithm to another. The modularity that this approach brings allows defining new algorithms without having to change the architecture. Figure 27 shows the architecture that was defined.

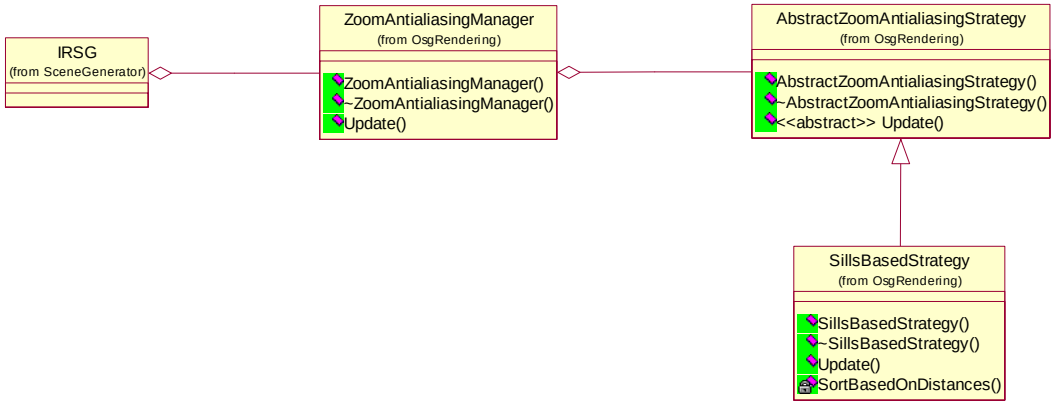

Figure 27: Zoom antialiasing capacity as a strategy pattern.

- `IRSG` is the module which uses the zoom antialiasing capacity;
- `ZoomAntialiasingManager` controls the creation of strategies and which strategy shall be used;
- `AbstractZoomAntialiasingStrategy` is a base class from which each ZAA strategy needs to inherit from; and
- `SillsBasedStrategy` is the current developed strategy based on the works of [9].

6.2.2 Multi-pass rendering

A multi-pass capacity is necessary as it allows the use of multiple fragment shaders, one after the other, to generate a single frame. For example, it is possible to use multiple shaders to downsample a texture instead of using only one. Figure 28 depicts a generic view of what is currently possible with the capacity: during a render stage, a texture is consecutively modified by a series of shaders.

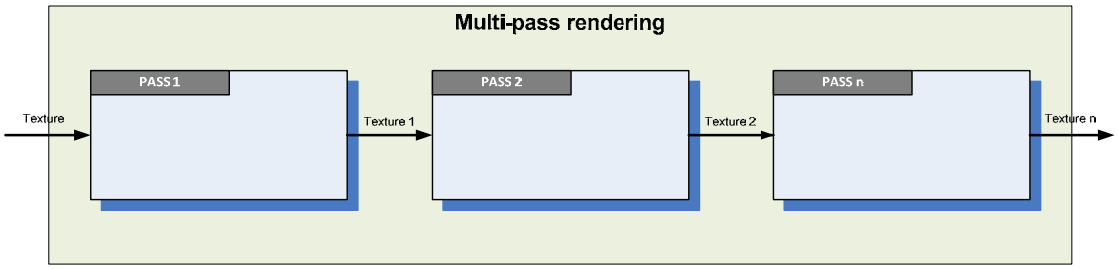

Figure 28: Generic multi-pass view.

6.2.2.1 Pass aggregator

`MultipassComposer` is a class responsible to create the pipeline of passes that a texture must pass through during rendering. A render to texture pass can be used to generate the first texture which will be used as input of the following pass. At this moment, downsampling passes can be used in the next passes to reduce the size of the texture.

6.2.2.1.1 Render to texture pass

The render to texture (RTT) technique, also called off-screen rendering, allows rendering a scene into a texture i.e. a buffer is copied into a texture. The frame buffer object (FBO) extension allows RTT in a platform-independent way. RTT is commonly used to implement a variety of image filters and post-processing effects by capturing images that would normally be drawn to the screen. `RttPass` can be used to accomplish this task.

6.2.2.1.2 Downsampling pass

The class `DownsamplingPass` is used to downsample an input texture by a factor 2 or 4.

6.2.3 Utilities

Some utility functions, regrouped in the class `UtilityFunctions`, were also developed to help manipulating OSG objects.

It allows:

- quads manipulation to receive a texture;
- cameras manipulation, to control field of view, position/rotation. It is also possible to create a geometry which represents the frustum of the camera (see Figure 29). This is purely a feature to help the debugging process, to see the zoom camera's frustum compared to the main camera's frustum for instance (as previously show in Figure 13);

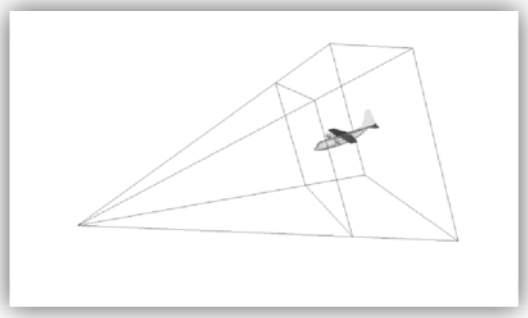

Figure 29: Camera's frustum representation.

- bounding volumes representation which can help validating the overall zoom antialiasing effort (see Figure 30).

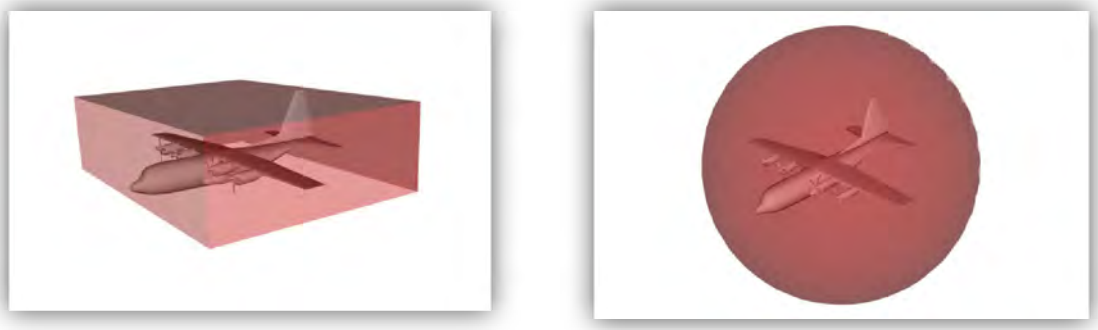

Figure 30: Bounding box (left) and bounding sphere (right) representation of a 3D model.

6.2.4 Object's size based level-of-detail

`PixelSizeBasedLod` is a class which inherits from `osg::LOD`. The process to switch from an object to another one, when the first object reaches a certain size, is already defined in the superclass. `PixelSizeBasedLod` simply:

- keeps the currently calculated size in a member variable available via an accessor;
- writes a message in the console when a switch occurs (from a 3D model to the quad; and vice versa).

7 Background

Prior to this contract, the background representation was really simplistic. A uniform background was assumed based on a spectrum value defined in the `Environment` parameters file: the resulting color, computed by the `SceneGenerator3D`, was associated to each pixel of the background, without considerations of the altitude, the LOS, etc. Thus, to enhance the background representation, various mechanisms were developed. The first step was to implement variations of the background value according to the value returned by the atmospheric model for given sensor altitude and LOS. The apparent background radiance is then computed as:

$$L_{app}^{bkgr} = \int L_{bkgr}(\lambda) R_{sens}(\lambda) \mathrm{d}\lambda . \tag{4}$$

In the following sections, the definition of a non-uniform background, created from samples taken at multiple LOS within a frame, is presented. The blending of textures with this background to create clutter effects is also documented.

7.1 Multiples background values

A uniform background is now defined using a single value taken in the middle of a sensor field of view and applied to each pixel of the background. A non-uniform background is created by obtaining multiple values from different LOS and by placing the values at the appropriate position on the background. An interpolation between those values is done to fill the remaining space between the calculated values. Figure 31 shows the two mechanisms that can be used.

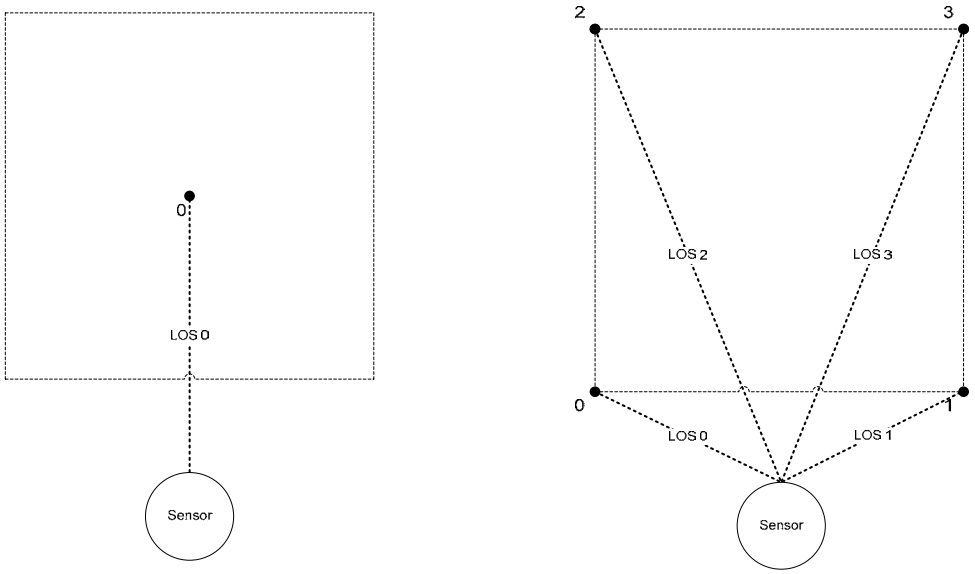

Figure 31: Single background value (from 1 LOS) vs. multiple values (from 4 LOS).

7.1.1 Background geometry

The background geometry is defined with a series of `GL_QUAD_STRIP`. The apparent background radiances are associated to the vertices of this geometry, and OpenGL interpolation is performed between the vertices.

A `QUAD_STRIP` is a group of connected quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data. Figure 32 shows the difference between the `GL_QUADS` and `GL_QUAD_STRIP`.

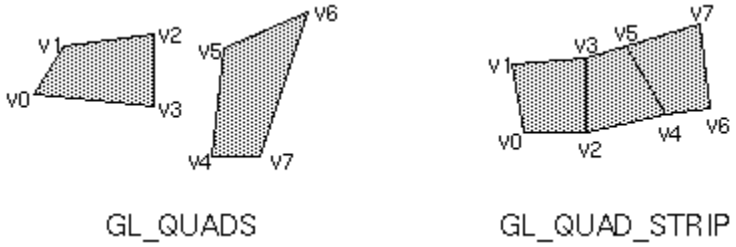

Figure 32: GL_QUADS vs. GL_QUAD_STRIP.

Figure 33 shows how the background geometry is constructed with `GL_QUADS_STRIP` and the result of the interpolation.

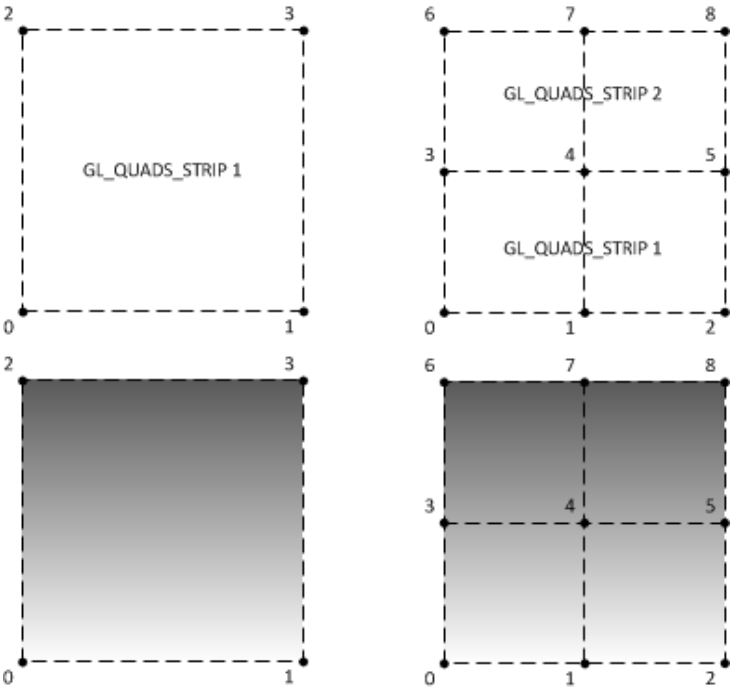

Figure 33: Interpolation within the QUAD_STRIP.

The geometry has a color binding property that needs to be set with `BIND_PER_VERTEX` to be able to use a different color (radiance value) for each vertex. The vertex and fragment shaders presented in Table 12 and Table 13 are also required to get interpolated colors between those vertices. When a constant background is used, the color binding is set to `BIND_OVERALL`. It tells OpenGL to consider only one color per geometry instead of per vertex.

Table 12: Vertex shader for the background geometry.

```
varying vec4 vertexColor;

void main(void)
{
    vertexColor = gl_Color;
    gl_Position = ftransform();
}
```

Table 13: Fragment shader for the background geometry.

```
varying vec4 vertexColor;

void main(void)
{
    gl_FragColor = vertexColor;
}
```

In KARMA simulations, the parameters file of an `ImagingSensor` allows to control the uniform/non-uniform background, via the `IsUniformBackground` parameter. To define a non-uniform background, the parameters `NumberOfBackgroundColumns` and `NumberOfBackgroundRows` shall be used.

In SMAT, the *Options* dialog box, shown in Figure 34, allows to control the activation/deactivation of the non-uniform background. If the non-uniform background is selected, the user can enter the number of rows and columns to be used to represent the background.

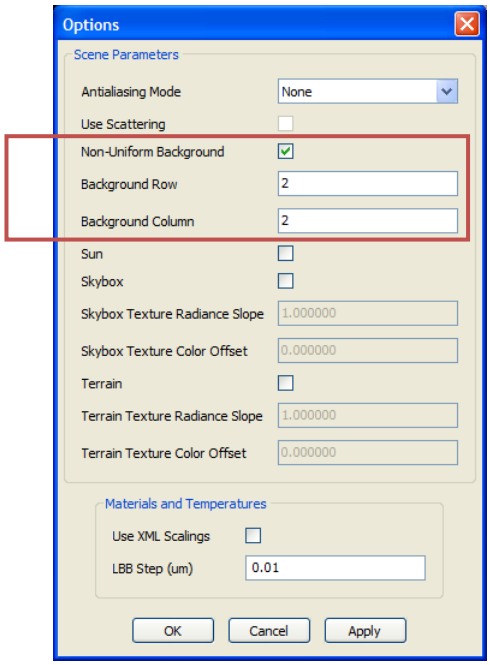

Figure 34: Using the non-uniform background in SMAT.

7.1.2 Using SMART to obtain background values

An important consideration with the non-uniform background is that SMART shall be used to obtain different values for each LOS. For KARMA simulations, if the `AtmosphereSmart` model is not in the `Environment` composition, a default behaviour will take place. This behaviour depends on the calculation mode of the `ImagingSensor` model (see `CalculationMode` parameter).

- In spectral mode, a default background value will be returned by the `Environment` based on the spectrum defined in its parameters file (see `BackgroundRadianceWavelengths` and `BackgroundRadianceValues` parameters);

- In wideband-ck mode, a default background value will be returned by the `Environment` (`NULL` is returned).

Inside SMAT, the parameters of an analysis allow selecting SMART to compute the background radiance, as shown in Figure 35, in order to have a non-uniform background (if this feature is activated in SMAT *Options* dialog box).

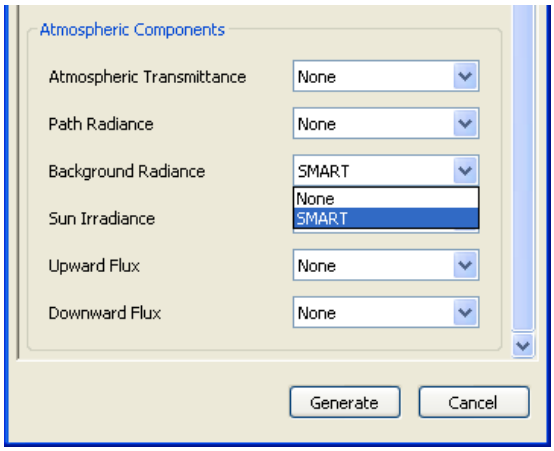

Figure 35: Using SMART to calculate background radiance in SMAT.

7.2 Sky and terrain textures

The multiple point background presented above can reproduce realistic values of background radiance within a frame. However, it may not be sufficient when sources of clutter, such as clouds or land, need to be represented. As presented below, the previously discussed background can then be blended with sky and land textures to further increase the scene realism.

7.2.1 Skybox

In video games, the background is often simulated by encapsulating the game level into a cube with carefully chosen textures on each face. These textures are used to represent distant objects such as the sky, mountains or unreachable buildings to give the impression that the background is infinite. Moreover, it is common for the skybox to remain stationary with respect to the viewer. This technique enforce the illusion of being very far away since other objects in the scene appear to move, while the skybox does not. Figure 36 shows an example of skybox.

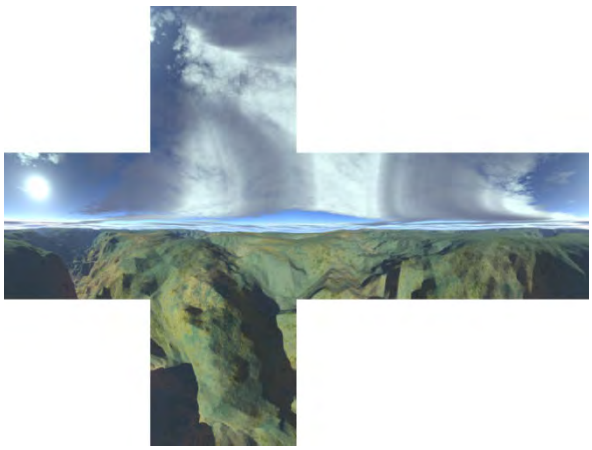

Figure 36: An example of skybox.

Notice that the skybox texture shown above includes a sun. It would be preferable to not have this artefact embedded in the texture since the sun is already modeled has an entity in the scene (more details are presented below). Otherwise, the sun located on the skybox texture would appear with wrong position and radiance on the images generated by the IRSG.

7.2.2 Terrain geometry

The skybox gives a good realistic effect when the sensor FOV contains sky and far away land only. However, as the skybox remains stationary with respect to the sensor (i.e. follows the sensor), the skybox is not suited to reproduce the relative motion of terrain seen from a sensor. Whenever sources of moving clutter are needed, a fully textured terrain geometry representing a real terrain can be used as shown in Figure 37. The terrain geometry does not remain stationary with respect to sensor as the skybox does; it rather uses a fixed position in the scene.

Figure 37: An example of terrain geometry.

7.2.3 Skybox and terrain in IRSG

The fragment shader shown in Table 14 is used to scale each pixels of the original texture (for both skybox and terrain) before being combined with background radiance pixels as detailed in Section 7.4. The color to radiance scaling can be customized with the `textureRadianceSlope` and `textureColorOffset` parameters. `textureRadianceSlope` represents the variation of radiance associated with a variation of 1 (from pure black to pure white) in texture color, while `textureColorOffset` is the texture color value (between 0 to 1) producing no variation in background radiance. Texture values above `textureColorOffset` are additive, while values below are subtractive.

Table 14: Fragment shader for the skybox and terrain.

```
uniform sampler2D texture;
uniform float textureRadianceSlope;
uniform float textureColorOffset;

void main (void)
```

```
{
    vec4 textureColor = texture2D(texture,  gl_TexCoord[0].st);
    float finalRed = textureRadianceSlope * ( textureColor.r - textureColorOffset);
    gl_FragColor = vec4(finalRed, 0.0, 0.0, 1.0);
}
```

For KARMA simulations, the `textureRadianceSlope` and `textureColorOffset` parameters (for the skybox and the terrain) are defined for each `ImagingSensor`, in their parameters file. Two booleans allows controlling the activation of these features: `UseSkyboxBackground` and `UseTerrainBackground`. The skybox textures and terrain already defined in `Environment` are also reused.

In the case of SMAT, Figure 38 shows where the skybox and terrain parameters can be modified in the *Options* dialog box. The terrain is currently hardcoded to use the following model: `$(KARMA_ROOT)\Utilities\ViewerDelta3d\3dModels\Releasable\Terrain\ArabTown.k3d`. For the skybox, the following textures are used:

- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\S.jpg;`
- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\N.jpg;`
- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\W.jpg;`
- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\E.jpg;`
- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\Up.jpg;` and
- `$(KARMA_ROOT)\\Utilities\\SMAT\\skybox\\current\\Down.jpg`.

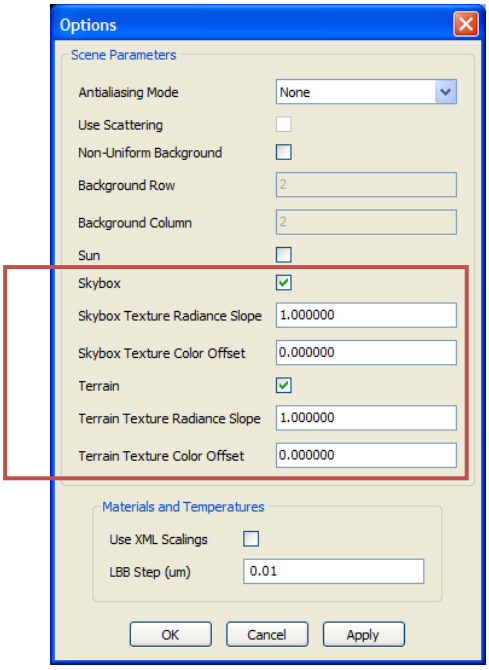

Figure 38: Using a skybox and terrain to model the background in SMAT.

7.3 Solar disc in IRSG

The solar disc is implemented as a 3D sphere into the scene. It always appears with a 0.53 degree diameter in the sensor's FOV. Like the skybox, the solar disc remains stationary with respect to the sensor. Its position relative to the sensor is set using sun azimuth and elevation defined in `Environment`. Its apparent radiance is given by

$$L_{appsun} = \frac{1}{\Omega_{sun}} \int E_{sun}(\lambda) R(\lambda)\, d\lambda, \tag{5}$$

where Ω_{sun} is a constant equal to 6.72×10^{-5} (with 0.53^{o} angular size), $E_{sun}(\lambda)$ is the sun irradiance at sensor's position (returned by the atmosphere model), and $R(\lambda)$ is the sensor's spectral response. The simple fragment shader shown in Table 15 is used to set the color of the sun.

The sun disc is always activated in KARMA simulations. Figure 39 shows the activation parameter for the sun disc inside the *Options* dialog box in SMAT.

Table 15: Fragment shader for the sun.

```
uniform float sunRadiance;

void main (void)
{
    gl_FragColor = vec4(sunRadiance, 0.0, 0.0, 1.0);
}
```

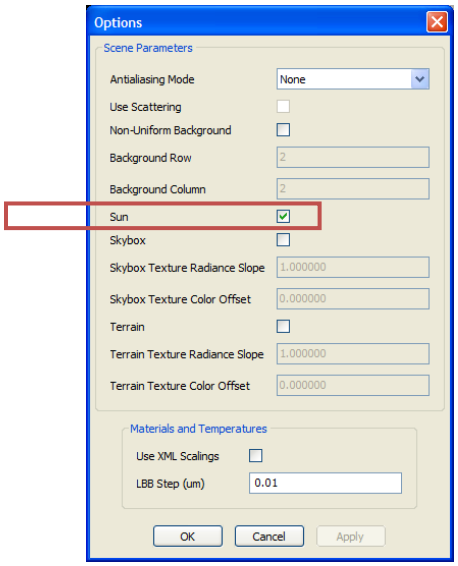

Figure 39: Activating the solar disc in SMAT.

7.4 Rendering

In order to achieve a realistic effect, the background geometry needs to be blended with the skybox and the terrain. However, the terrain needs to occlude the skybox without any blending between these two geometries. To achieve this, both are rendered opaquely before being blended (additive) with the background geometry (defined in section 7.1.1).

7.4.1 Pre-render camera

OSG offers multiple mechanisms to control rendering order of geometries. Since the background geometry needs to be rendered over the entire FOV, it needs to be rendered using an orthogonal projection matrix, associated with a pre-render camera (it cannot be done using a single camera). This kind of feature is widely used in video games when rendering heads-up display (HUD) over the game scene.

As shown in Table 16, two pre-render cameras are needed to control the rendering order before the main camera renders the scene. The first camera renders the skybox, the sun and the terrain (in this order) using a perspective projection matrix into the texture. The next camera renders the background geometry over the same texture using an additive blending function. This blends the radiance variations caused by sun, skybox, and terrain with the radiance from the atmosphere models (associated to the background geometry).

Table 17 details the equation of the background geometry blending function. Figure 40 shows how a non-uniform background, i.e. the multiple background values computed using SMART, is combined with the skybox.

Table 16: Using two pre-render cameras before the main camera.

```
// draw the background first
m_backgroundCamera->setRenderOrder(osg::Camera::PRE_RENDER, 1);
m_environmentCamera->setRenderOrder(osg::Camera::PRE_RENDER);

// Background and environment camera both render into the FBO
m_backgroundCamera->setRenderTargetImplementation(osg::Camera::FRAME_BUFFER_OBJECT);
m_backgroundCamera->attach(osg::Camera::COLOR_BUFFER0,
m_offscreenTexture.get(),0,0,false,0,0);

m_environmentCamera->setRenderTargetImplementation(osg::Camera::FRAME_BUFFER_OBJECT);
m_environmentCamera->attach(osg::Camera::COLOR_BUFFER0,
m_offscreenTexture.get(),0,0,false,0,0);

// Add the camera to the viewer
m_viewer->addSlave(m_backgroundCamera, false);
m_viewer->addSlave(m_environmentCamera, false);

// The GL_COLOR_BUFFER_BIT MUST be cleared with opaque black
m_environmentCamera->setClearColor(osg::Vec4f(0.0f,0.0f,0.0f,1.0f));

// The GL_COLOR_BUFFER_BIT must NOT be cleared
m_backgroundCamera->setClearMask(GL_DEPTH_BUFFER_BIT);
m_viewer->getCamera()->setClearMask(GL_DEPTH_BUFFER_BIT);
```

Table 17: Blending function when rendering the background geometry.

Destination factor = $(D_r, D_g, D_b, D_a) = (1, 1, 1, 1)$ and source factor = $(S_r, S_g, S_b, S_a) = (1, 1, 1, 1)$ $(R, G, B, A) = (R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a)$ $(R, G, B, A) = (R_s(1) + R_d(1), G_s(1) + G_d(1), B_s(1) + B_d(1), A_s(1) + A_d(1))$ $(R, G, B, A) = (R_s+R_d, G_s+G_d, B_s+B_d, A_s+A_d)$

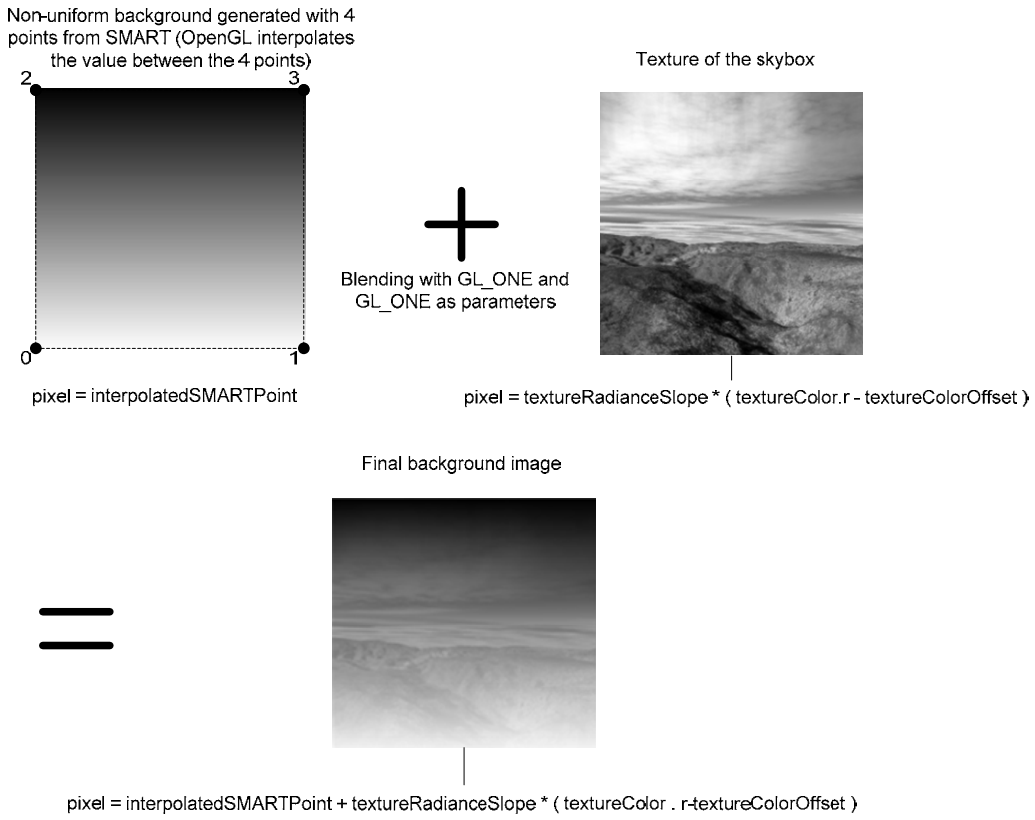

Figure 40: Processing of the final background image.

7.5 KARMA architecture

Figure 41 and the following text show the modifications that were done to the KARMA architecture to make possible the changes discussed previously in this section of the report.

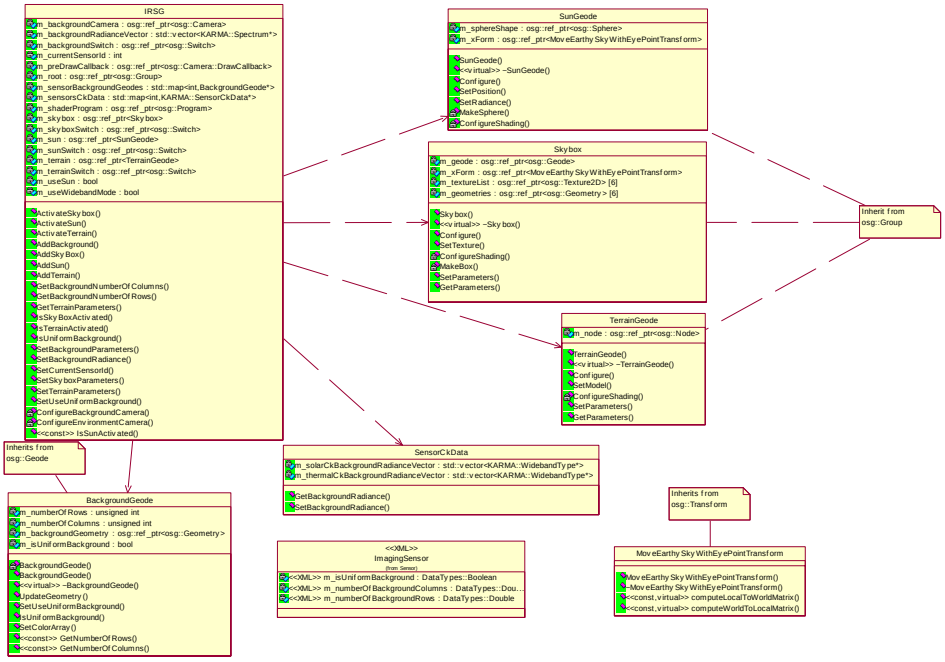

Figure 41: Non-uniform background class diagram.

- The `BackgroundGeode` class encapsulates a background geometry used to provide a uniform or non-uniform background in the IRSG. When the background is uniform, one color (radiance) is assigned to the whole geometry while one color is assigned to each vertex when the background is non-uniform.

- The `Skybox` class encapsulates the skybox. The skybox is parameterized using six textures and two `uniform` variables.

- The `TerrainGeode` class encapsulates the terrain geometry. The terrain is parameterized using a 3D model and two uniform variables.

- The `SunGeode` class encapsulates the solar disc geometry. No texture is required since the geometry is uniformly colored dynamically using sun radiance at the sensor's location.

- The `MoveEarthSkyWithEyePointTransform` class contains the transformation to allow a `osg::Node` to follow the eye point (sensor). This is used by the skybox and the sun to give the impression that they are located at an "infinite" distance.

- The `IRSG` class handles background camera, background switch, skybox activation and served as interface to other classes (`BackgroundGeode`, `Skybox` and `SensorCkData`).

- The `ImagingSensor` contains the background and skybox parameters to allow a configuration for each sensor.

8 Database properties

For each model which participates in the scene generation process, a database file is associated to the 3D model file, and contains the infrared properties. In this section, the modifications that were made to the database are presented.

8.1 User defined spectrum

The signature of each model is characterized by materials and temperature properties, associated to material and temperature indexes in the 3D model. These features can be edited by the way of SMAT database editor dialog. Initially, the temperature was defined using an equation and then used in the Planck equation to compute radiance. A new approach called user defined spectrum has been added, allowing to define surface radiance using a combination of one or more radiance spectrums. These spectrums are called components. When this mode is used, the thermally emitted contribution to the apparent radiance as seen through the atmosphere is given, as opposed to equation (1), by

$$L_{app}^{therm} = \varepsilon_{scaling} \cos^{N}(\theta) \int \sum_{j} c_j(t,\phi) L_j(\lambda) \tau_{path}(z_0,\lambda) R_{sens}(\lambda) \mathrm{d}\lambda, \tag{6}$$

where $L_j(\lambda)$ is the radiance spectrum of a component of index j and $c_j(t,\phi)$ is its weighting factor according to the entity's time and the angle between the entity's orientation vector and the view vector.

Some changes in SMAT database dialog editor window were necessary to support the user defined spectrum mode. The first change was to add a new tab to manage user defined spectrum importation. Figure 42 shows the tab in the database editor dialog to import a set of user defined spectrums.

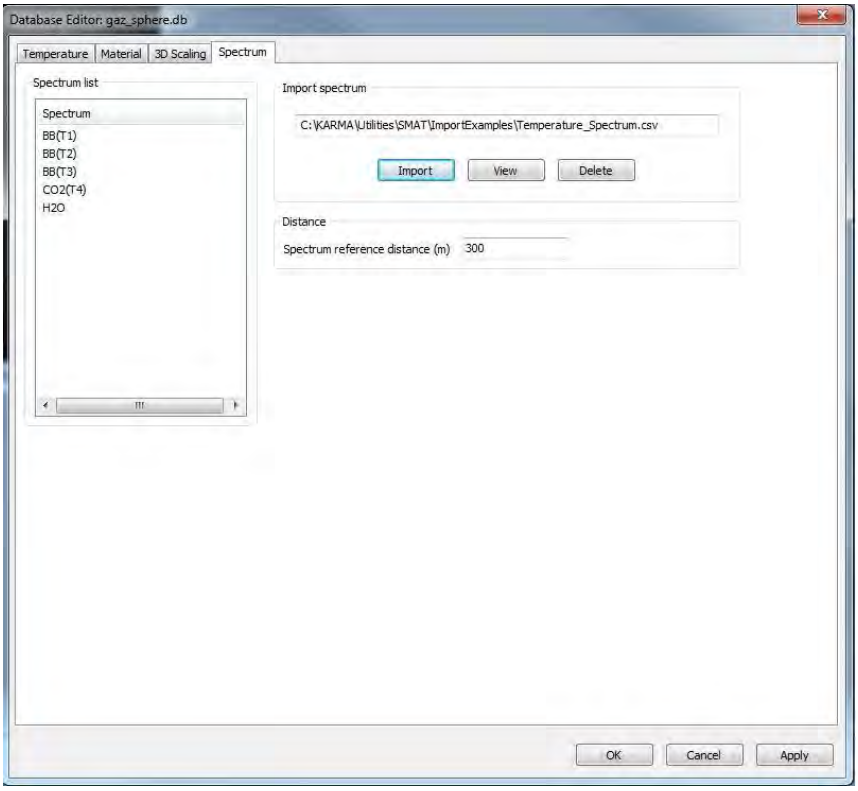

Figure 42: User defined spectrum import tab.

- The user defined spectrums are imported from a CSV file by using the *Import* button. The file is read and stored in the database file of a given signature model. A list of spectrum is displayed in the list on the left side of the tab.

- It is possible to view imported data in table format by using the *View* button.

- The *Delete* button can be used to clear imported spectrums.

- The *Spectrum reference distance* is used to specify the distance used to measure the user defined spectrum (i.e. distance from the object).

It is necessary to take into account *Spectrum reference distance* in atmospheric transmittance computation. As shown on Figure 43 two situations can occur. In the first situation (A), the sensor is within the reference distance (nearer), thus no additional transmittance is computed since it is already taken into account in the user defined spectrum. In the second situation (B) the sensor is outside the reference distance (farter), thus the atmospheric transmittance will be computed using the sensor distance minus the *Spectrum reference distance* (green distance).

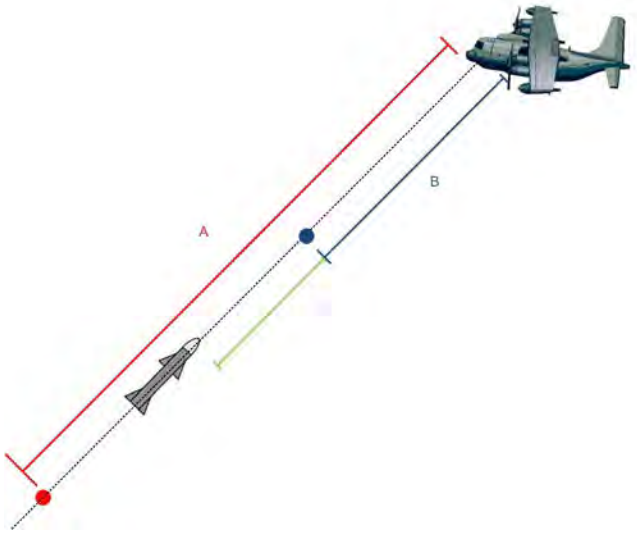

Figure 43 : Spectrum reference distance.

8.1.1 User defined spectrum file format

The user defined spectrums are stored in a comma separated values (CSV) format. This format stores a table that is easy to see and modify using a commercial spreadsheet tool such as MS Excel. Table 18 shows an example of the file format. The first column of this table is the frequency values of the spectrums; all spectrums, referred as component, share the same frequency values. The first cell of this column defines the units of the frequency values: the tag "um" defines that the spectrum is given in wavelength while the tag "cm-1" indicates a spectrum defined in wave number. Internally, the spectrum is stored in wavelength; therefore the input data read in wave number is converted to wavelength. The following columns define the values for each spectrum component. The first cell of each column defines the name of the component; this name will be displayed in spectrum list as shown earlier in Figure 42.

Table 18: User defined spectrum file format example.

cm-1	BB(T1)	BB(T2)	BB(T3)	CO2(T4)	H2O
1900	7.76E-05	3.61E-05	2.24E-05	4.55E-05	0.000159
1902	6.75E-05	3.15E-05	1.96E-05	5.00E-05	0.000134
1904	4.53E-05	2.13E-05	1.32E-05	5.43E-05	9.03E-05
1906	2.08E-05	9.96E-06	6.28E-06	5.25E-05	4.59E-05
1908	1.46E-05	7.11E-06	4.53E-06	4.46E-05	3.26E-05
…	…	…	…	…	…

8.1.2 Temperature properties

The temperature tab of the database editor dialog has been modified to add the user defined spectrum mode settings. The *Mode* property has been added to allow selecting one mode or another. The former approach based on a temperature equation is still available using the *Use Temperature* mode as shown in Figure 44. The equation parameters have been grouped in a tab.

The properties of the selected mode are shown in a tab section to distinguish the properties of this mode and the user defined spectrum mode.

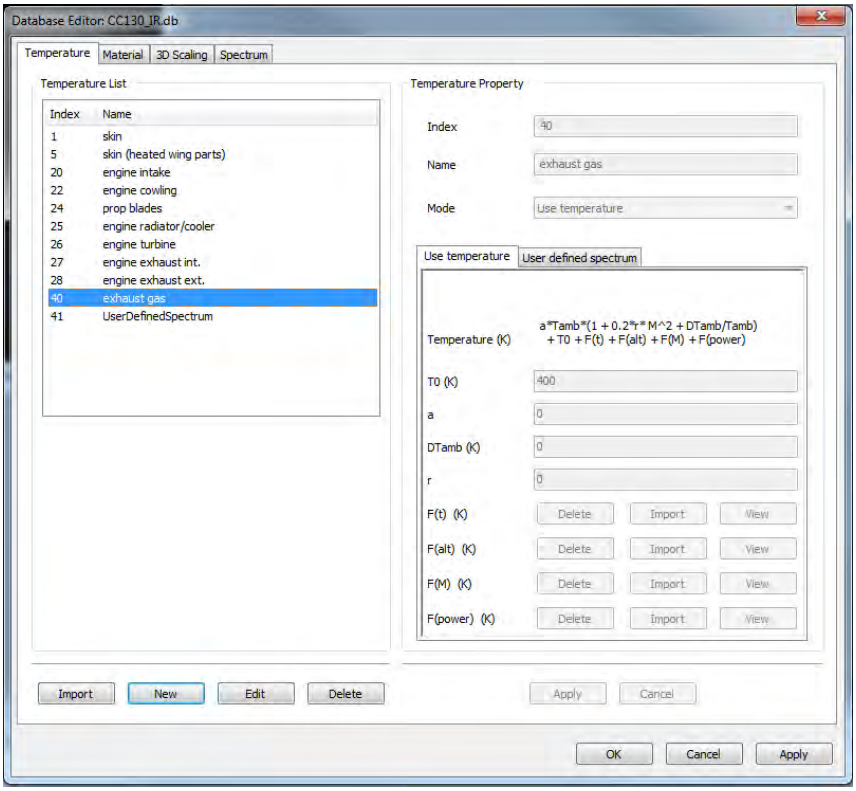

Figure 44: Temperature tab with use temperature mode.

Figure 45 shows the dialog in user defined spectrum mode. This mode allows the user to define a radiance spectrum using a combination of components (e.g. CO2, H2O) defined in the *Spectrum* tab. Notice that user defined spectrums must be imported as presented previously before using the user defined spectrum mode.

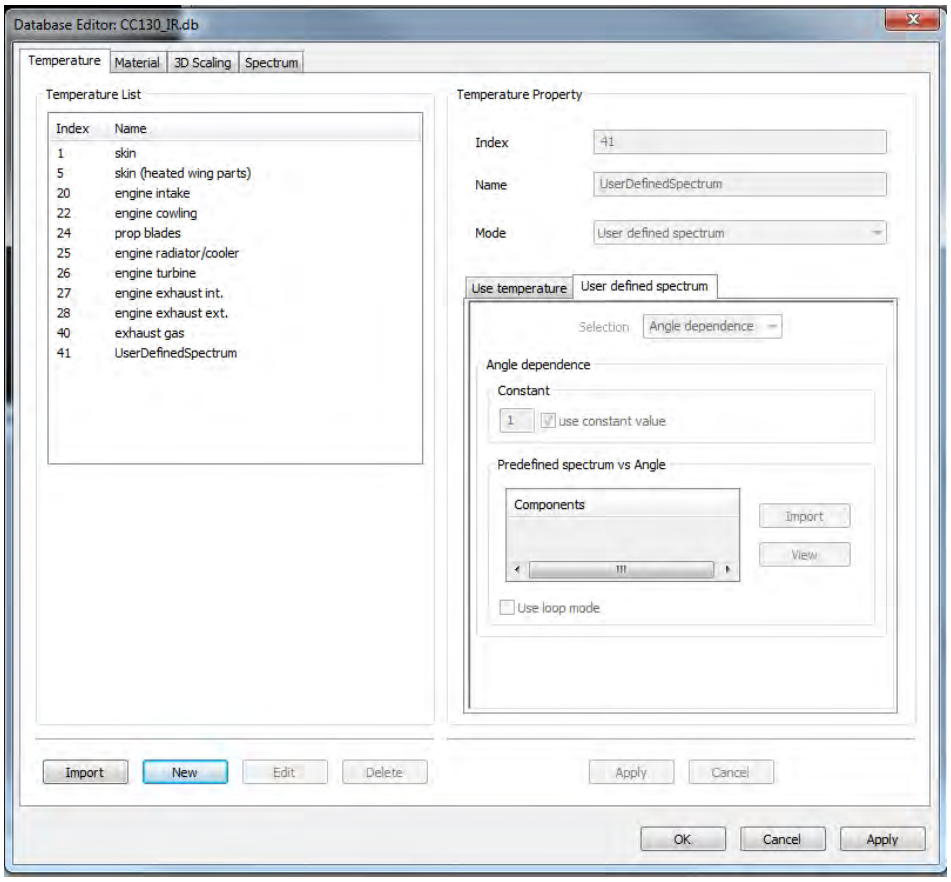

Figure 45: Temperature tab with user defined spectrum mode.

The user can select angle and time factors to apply on each component of the spectrum using the *Selection* property. These factors can be defined with a constant or with a lookup table.

- The lookup tables are imported from a CSV file by using the *Import* button. The file is read and stored in the temperature properties and a list of components is displayed in the list in the middle of the tab.

- It is possible to view imported data in table format by using the *View* button.

- The *Time dependence* lookup table allows the user to define temporal evolution of the factor for each component of the spectrum. The *Use loop mode* checkbox allows to define periodic time lookup table.

- The *Angle dependence* lookup table allows the user to define angular evolution of the factor for each component of the spectrum.

The lookup tables are stored in CSV format. Table 19 and Table 20 show an example of the file format for time and angle factor. The first column of the *Angle dependence* lookup table is the angle defined in degrees while the first column of the *Time dependence* lookup table is the

entity's time defined in seconds. For both lookup tables, the first cell of this column should be empty. The following columns contain factors for each spectrum component. The lookup tables must use spectrums, using their name, already imported in the *User defined spectrum* tab.

Table 19: Time dependence lookup table file format example.

	BB(T1)	BB(T2)	BB(T3)	CO2(T4)	H2O
0	0.5	0.7	0.1	0.1	0.1
1	0.6	0.8	0.2	0.2	0.2
2	0.7	0.9	0.3	0.3	0.3
3	0.8	1.0	0.4	0.4	0.4
4	0.9	1.0	0.5	0.5	0.5
…	…	…	…	…	…

Table 20: Angle dependence lookup table file format example.

	BB(T1)	BB(T2)	BB(T3)	CO2(T4)	H2O
0	0.3	0.4	0.1	0.7	1.0
45	0.4	0.5	0.2	0.8	1.0
90	0.5	0.6	0.3	0.9	1.0
135	0.6	0.7	0.4	1.0	1.0
180	0.7	0.8	0.5	1.0	1.0
…	…	…	…	…	…

8.2 Temperature lookup tables

The temperature equation used in the Planck equation is now referred as *Use temperature* mode. Using this mode, the surface temperature is defined using an equation, as shown previously in Figure 44. Additional contributions have been implemented to allow modelling the temperature according to the entity's conditions: its altitude, its speed and its power. The surface temperature is given by

$$T_{surf} = T_0 + aT_{amb}\left[1 + 0.2rM^2 + \frac{\Delta T_{amb}}{T_{amb}}\right] + f(t) + f(power) + f(alt) + f(M), \quad (7)$$

where *f(power)*, *f(alt)* and *f(M)* are the power, altitude and speed lookup tables respectively.

8.3 N angle factor

The emissivity of a surface is defined spectrally in the material tab of the database editor dialog. This tab has been modified to include a property named *N angle factor*, as shown in Figure 46, which is used to reproduce emissive materials with either spatially uniform or spatially varying radiance. This property is used in the fragment shader presented in Section 4 to modulate the thermal radiance according to the view angle using the following factor: $|\cos^N(\theta_{view})|$, where **N** is the *N angle factor* and θ_{view} is the angle between the view vector and the surface's normal. The factor is applied to the emissive component in both temperature and user defined spectrum modes, as shown in equations (1) and (6). Figure 47 shows the resulting factor as a function of the

view angle for different *N angle factor*. Therefore, a spatially uniform radiance is obtained when the *N angle factor* is 0 as the modulation factor is always 1 no matter the view angle.

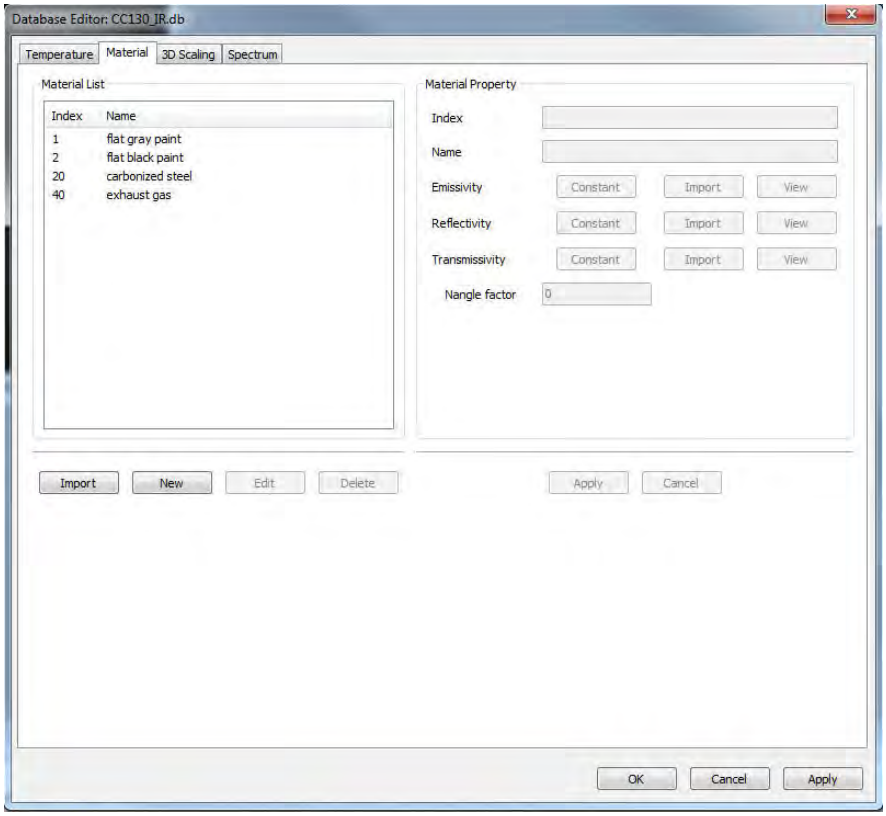

Figure 46: Material tab with the N angle factor.

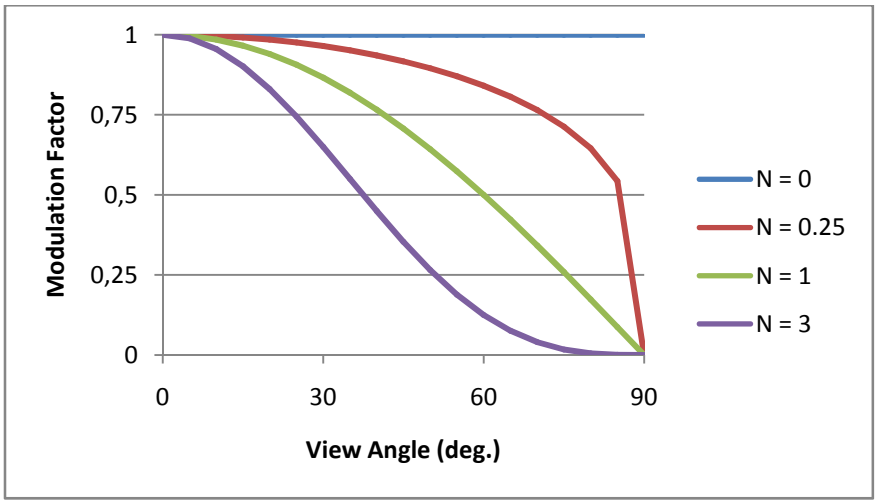

Figure 47: Modulation factor computed according to the view angle and for different N angle factors.

9 Scattering

As a consequence of light scattering on aerosols, the atmosphere degrades images in terms of light intensity and sharpness. For example, an object that appears as a black and white chessboard, when seen at a certain distance, will eventually appear as a homogeneous gray square if it is seen at a farther distance. Figure 48 [10] shows contrasts fading as mountains are farther from the observer. This effect has been included in the IRSG by applying a filter on images. The filter is based on the modulation transfer function (MTF), i.e. the point spread function (PSF) in the frequency domain.

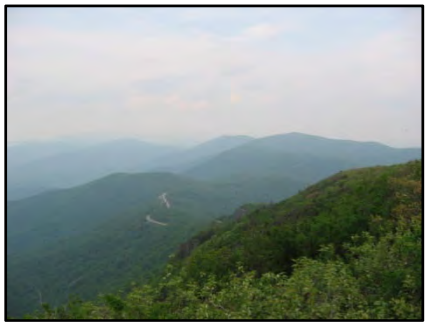

Figure 48: Image degraded by the atmosphere.

9.1 MTF database

The stratified model used to generate the scattering induced MTF has been developed by Tremblay et al. and is explained in [11]. Figure 49 presents a reproduction of Figure 5 from [11]. It presents eight curves of the MTF as a function of the optical depth (OD) τ. This model has been used to create a MTF database to gather the variation of MTF as a function of optical depth for specific atmospheric conditions. A Matlab function has been used to write the MTF database binary file. The file format is described in Table 21. The first part of the file is the header. It contains MTF parameters than can be displayed to the user to identify the current database.

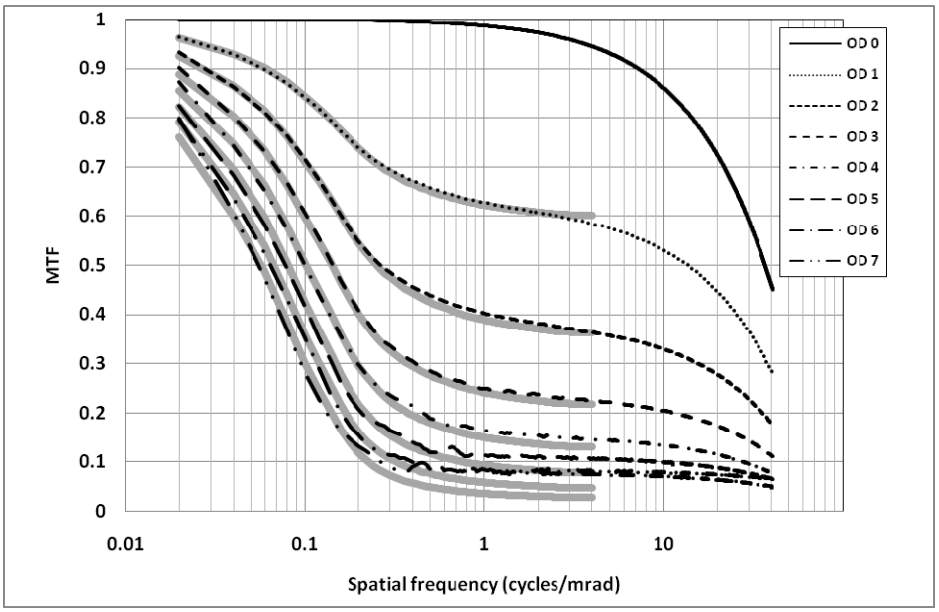

Figure 49: Reproduction of Figure 5 from [11]. "Comparison of MTF simulated with the Undique Monte Carlo simulator and the stratified model for water droplets 100 microns in diameter and 8 different optical depths. The gray lines show the stratified model results and the black superimposed lines show the Monte Carlo results" – the grey lines deviate from the dotted curves (Monte Carlo simulator) at high spatial frequencies since the optical system is included in the later.

Table 21: Description of the MTF binary format.

Type	Description
`double`	FOV
`double`	FOI
`double`	Particle diameter
`double`	frequency
`int`	Number of MTF
`int`	Number of points in the MTF
`double * Number of points`	Frequency values
For all MTF	
`double`	tau
`double * Number of points`	MTF values

For a given optical depth, the MTF is applied on an image as

$$L_{U+S}\left(\theta_x,\theta_y\right)=\left(\frac{P_U+P_S}{P_U}\right)TF^{-1}\left\{MTF\left(k\right)*TF\left\{L\left(\theta_x,\theta_y\right)\right\}\right\}. \tag{8}$$

The scaling factor before the inverse Fourier transform is necessary since atmospheric transmission is already taken into account by the IRSG. It is equivalent to

$$\frac{P_U+P_S}{P_U}=\frac{1}{MTF\left(\max\ freq\right)}, \tag{9}$$

where MTF(max freq) corresponds to the asymptotic MTF value of the stratified model at high spatial frequency. This scaling factor is directly included while populating the database. Hence, the MTFs are normalized to one at high frequency.

9.2 Using MTF for scattering

The MTF is used inside the IRSG to blur the image according to atmospheric scattering. The image is filtered at the end of the rendering step. It is necessary to apply the MTF individually on each object of the scene because the scattering is a function of the optical depth. For a given object, the optical depth is estimated by

$$\tau\approx-\ln\left\{\frac{\int T_{atm}\left(\lambda\right)R_{sensor}\left(\lambda\right)\mathrm{d}\lambda}{\int R_{sensor}\left(\lambda\right)\mathrm{d}\lambda}\right\}, \tag{10}$$

where T_{atm} is the atmosphere transmittance and R_{sensor} is the spectral response of the sensor.

To obtain one image for each object, the process uses the *OsgRendering* library (presented in Section 6.2), which replaces each 3D model by a quad in the scene. The quad size is near the size of the model's bounding sphere. However, for the scattering effect to take place appropriately, the 3D model shall be replaced by a quad having the same size (in pixels) as the sensor which is using the IRSG (Figure 50). Obviously, the range based LOD mechanism, as presented in Sections 6.1.1 and 6.2.4, is deactivated i.e. the quad is always displayed whatever the distance is between the platform and the sensor. An OSG post-draw callback is used to filter the texture which was applied on the quad with the MTF. The main disadvantage of this approach is that it is necessary to transfer texture data from the GPU to the CPU to perform texture filtering, which increases the process duration.

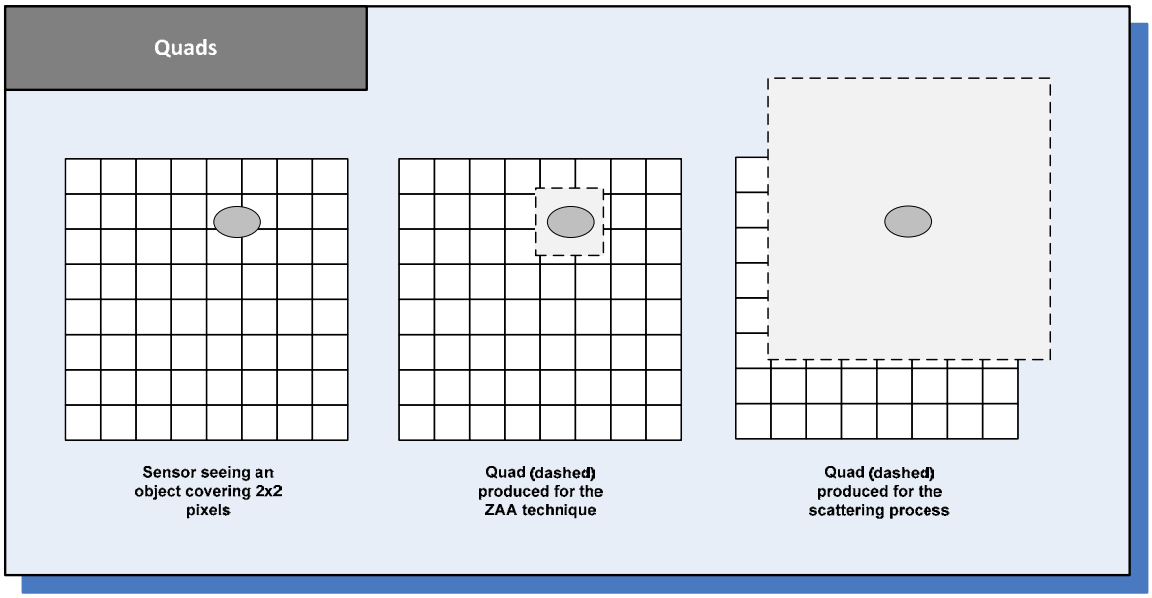

Figure 50: Different quads involved in different techniques.

The library FFTW is used to apply MTF on the resulting quad of each entity. This library is a free open source product in the context of non-profit product (GPL licence). A commercial version is also available and it is used in many commercial products like Matlab. The use of this library includes only a limited number of changes in KARMA solution. The files of FFTW have been added in the `%KARMA_ROOT%\Softwares\` directory.

Figure 51 presents a flowchart of major steps to apply MTF on the quad of each entity. The process is divided in three main operations. The first one is to get the data from the texture and copy values in a FFTW buffer. The second step is to apply the MTF on the image. Finally, the filtered data is taken from the FFTW buffer and put back in the texture.

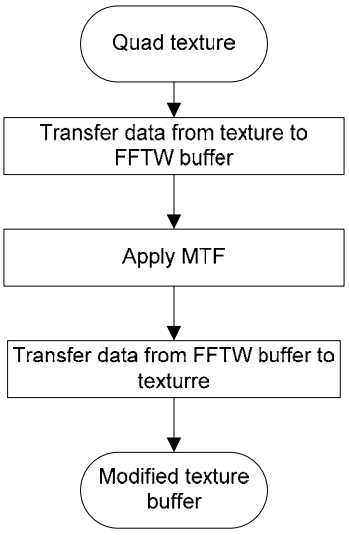

Figure 51: General process to apply MTF on the texture.

The `ApplyMTF` function is the central part of the algorithm. This function filters image with a given MTF.

The flowchart of the `ApplyMTF` function is shown at Figure 52. The MTF 1D, interpolated from the MTF database, is one of the two inputs of the function. It is necessary to transform the 1D MTF to 2D MTF according to image dimensions. The method is illustrated on Figure 53. The MTF computed by the model is a 1D curve. The filter buffer is filled by rotating the 1D MTF around central point. The MTF is applied only on the FOV of the sensor. The points outside the FOV are set to the last value of the MTF. The result of this operation is illustrated on Figure 54. This result is put in a 2D FFTW buffer ready for filtering operation.

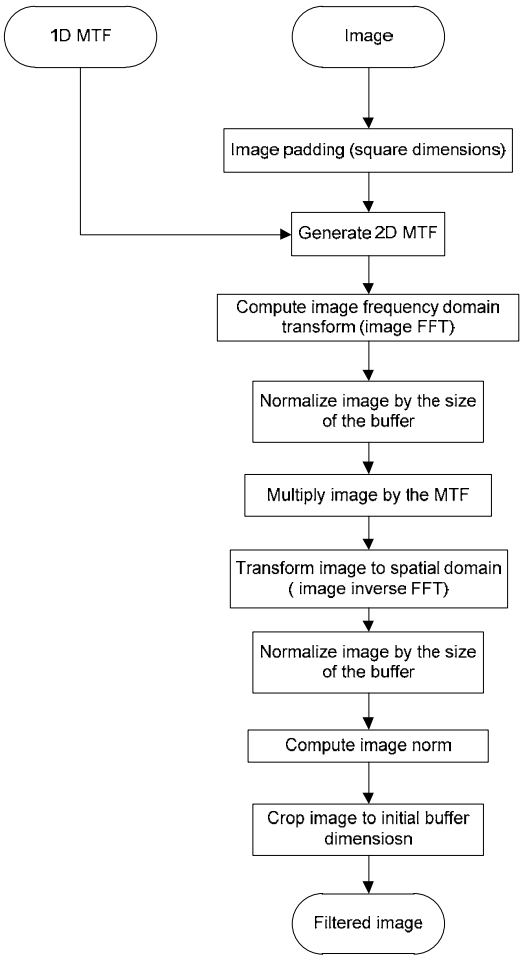

Figure 52: Flow chart of ApplyMTF function.

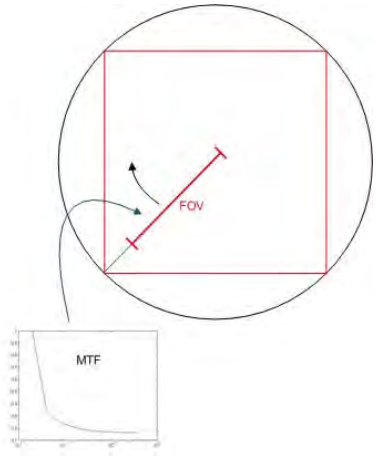

Figure 53: Method to create 2D MTF.

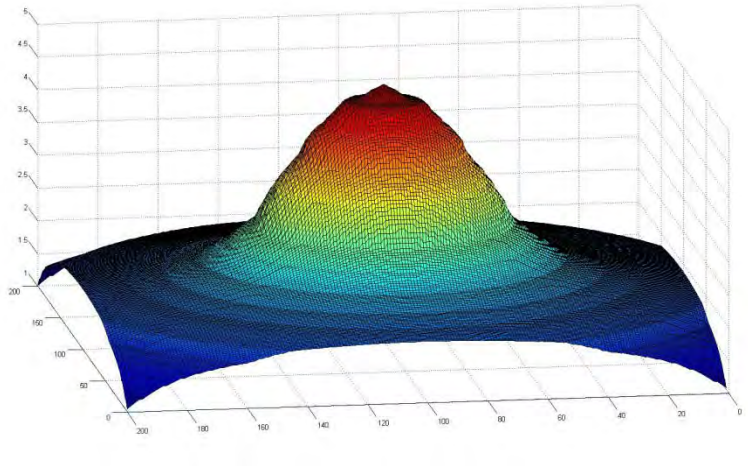

Figure 54: Example of 2D MTF.

The following steps apply the MTF filter on an image. The first step computes the image transform in frequency domain. This step is performed by FFTW. The second step is to apply the filter on the image by multiplying each buffer element to element. The third step is the inverse FFT transform computed by FFTW. The inverse transformation function of FFTW gives an unnormalized transform. It is necessary to normalize the result by the size of the buffer (number of rows * number of columns). Finally, the image is cropped to get a buffer that is the same size of the initial buffer.

Table 22: Apply MTF function.

```
void ApplyMTF(std::vector<std::pair <double, double>> &MTF, doubleFOV, int sizeX, int
              sizeY, fftw_complex **image)
{
    int sizeXInit = sizeX;
    int sizeYInit = sizeY;
```

```
    if (sizeX != sizeY)
    {
        PadImage (image, sizeX, sizeY);

        if (sizeX > sizeY)
            sizeY = sizeX;
        else
            sizeX = sizeY;
    }

    // Compute MTF
    fftw_complex *MTF2D = NULL;
    Generate2DMTF(MTF, sizeX, sizeY, FOV, &MTF2D);

    // compute FFT of the image
    fftw_complex *imageFFT = NULL;
    fftw_planplan;

    imageFFT = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sizeX * sizeY);

    plan = fftw_plan_dft_2d(sizeX, sizeY, *image, imageFFT, FFTW_FORWARD, FFTW_ESTIMATE);
    fftw_execute(plan);

    // shifting
    FFTShift(imageFFT, sizeX, sizeY);

    // multiply image by the mtf (frequency)
    for( int i = 0; i < sizeY*sizeX; ++i )
    {
        imageFFT[i][0] = imageFFT[i][0] * MTF2D[i][0];
        imageFFT[i][1] = imageFFT[i][1] * MTF2D[i][0];
    }

    FFTShift(imageFFT, sizeX, sizeY);

    // compute inverse transform
    fftw_destroy_plan(plan);
    plan=fftw_plan_dft_2d(sizeX, sizeY, imageFFT, *image, FFTW_BACKWARD, FFTW_ESTIMATE);
    fftw_execute(plan);

    FFTNormalize(*image, sizeX, sizeY);
    NormComplex (*image, sizeX, sizeY);

    //Crop image to get the original size
    CropImage(image, sizeX, sizeY, 0, 0, sizeXInit, sizeYInit);
    sizeX = sizeXInit;
    sizeY = sizeYInit;

    fftw_destroy_plan(plan);
    fftw_free (imageFFT);
}
```

9.3 Results

This section presents images produced by the process described in the previous section. Figure 55 presents images with and without the scattering effect on a simple sphere model. Notice that this is not the expected result. The image should rather show a diffuse circle rather than a cross. Due to the project ending, an investigation of the error leading to that result could not be conducted.

Figure 55: Image of the sphere model without (left) and with (right) scattering effect.

In KARMA simulations, the parameter `UseAtmosphereScattering` of an `ImagingSensor` allows to control the activation/deactivation of scattering. Notice that an atmospheric module (e.g. `AtmosphereScatteringLUT`), which can manage scattering requests, shall also be defined in the environment's composition. Figure 56 shows the scattering activation in the *Options* dialog box in SMAT. Notice that the zoom antialiasing (128, 256 or 512) must be activated in order to be able to use this feature in KARMA simulations and in SMAT.

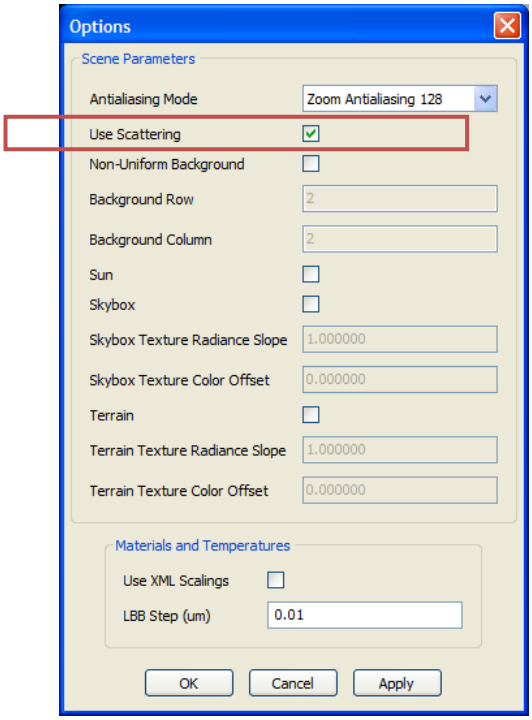

Figure 56: Activating the scattering in SMAT.

10 SMAT controls

During the project, some new controls and enhancements were done to SMAT. The following sections describe these new features.

10.1 Coordinate system

The view coordinate system was added at the left bottom side of the model viewer. The model coordinate system was also integrated and is located at the center of the model. Figure 57 shows these new additions. These coordinate systems can be shown or hidden via the *View* menu.

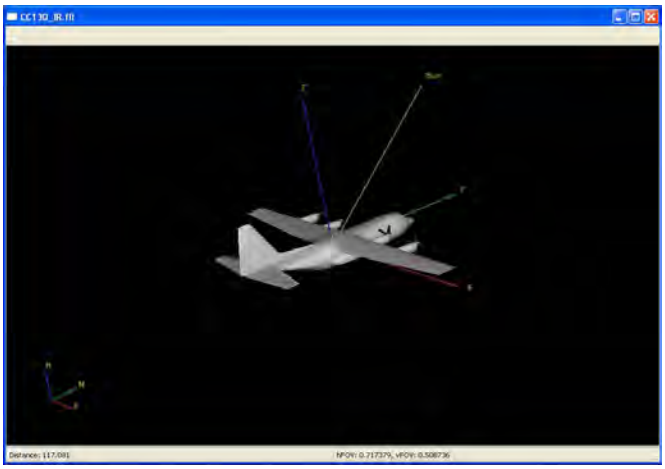

Figure 57: Model view inside SMAT.

10.2 Sun vector

The sun vector is represented by a yellow line originating from the center of the model and pointing towards the sun (see Figure 57). This vector can also be shown or hidden via the *View* menu.

10.3 Model-View manipulator

The manipulators were reviewed allowing a more intuitive and precise control over the scene. The camera is controlled with sliders, and now it is also possible to control the model's orientation. Such a control is required with the addition of a terrain and skybox in the scene. Values can also be entered in text fields, which allow inserting precise values (see Figure 58).

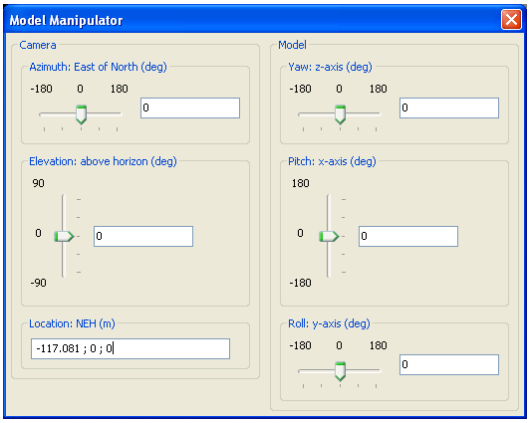

Figure 58: Camera and model manipulators inside SMAT.

10.4 Temperature profile

A new functionality was added to SMAT which allows generating a chart where the temperature is obtained from SMART for a range of altitudes (see Figure 59). A dialog box allows setting the analysis parameters (the minimum and maximum altitudes and the step) used to generate the chart.

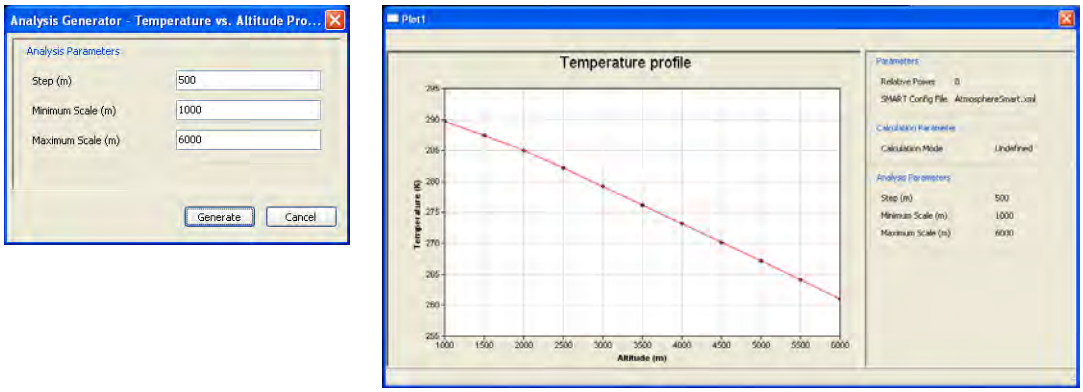

Figure 59: Setup and view a temperature profile.

10.5 Images comparison

A new tool in SMAT allows comparing images produced by SMAT or KARMA simulations in the CSV format. It is very useful to obtain the differences between two images to see, for example, the impact of an algorithm on the IRSG. Figure 60 shows an example where an image was produced without antialiasing (top left) and another one, with the ZAA 512 activated (top right). The image created from the comparison (bottom) shows where the differences are (around the propellers and the plumes).

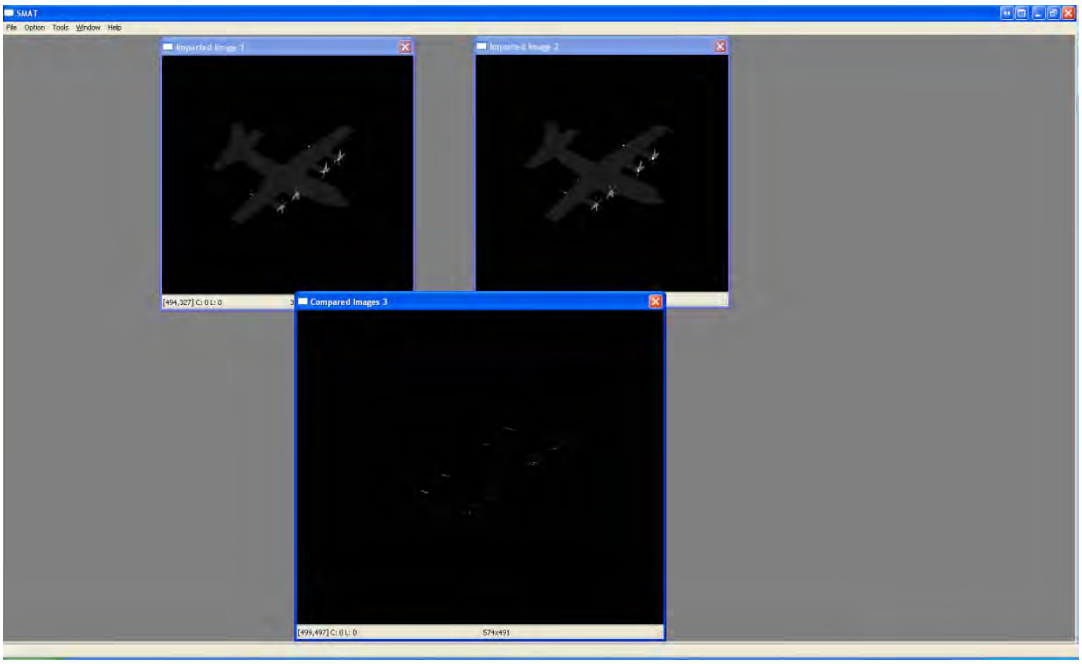

Figure 60: Comparing images with SMAT.

10.6 Polar plot

The polar plot analysis was reviewed to add a third execution mode. The first two modes were already developed but are still explained. Figure 61 shows the polar plot types available.

Figure 61: Polar plot analysis with SMAT.

10.6.1 Camera-Azimuth mode

The camera is placed on the horizontal plane, in front of the 3D model and looking at the center of the model. The camera will rotate around the model on the horizontal plane, around the scene Height axis (up axis); clockwise when viewed from the model. The view offset can be used to give an elevation to the camera.

10.6.2 Camera-Elevation mode

The camera is placed in front of the 3D model, in the horizontal plane of the model and looking at the center of the model. The camera will rotate counterclockwise around the X axis of the model. The camera begins its rotation by moving down.

10.6.3 Model-Yaw mode

The model pitch and roll are set to 0.0 and the model is rotated around its own Z axis (up axis) to face the camera. The camera is at the current position in the model viewer, and looking at the center of the model. The model rotates counterclockwise around its own Z axis.

10.7 Radiative outputs

A tool was added to SMAT to interrogate SMART (if SMART is correctly initialized) and obtain different spectrums that could be used into image-based analysis using the IRSG. This tool is accessible via the *Tools* menu, under *Radiative Output...* menu item as shown in Figure 62.

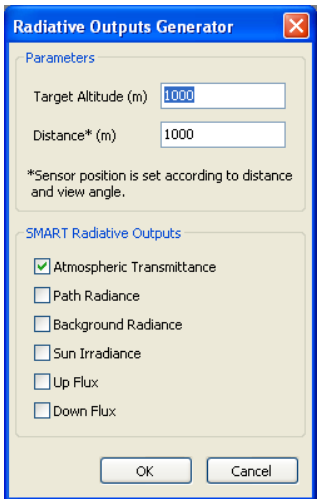

Figure 62: Radiative outputs generator inside SMAT.

It allows obtaining atmospheric transmittance, path radiance, background radiance, sun irradiance, up flux and down flux. The aspect of the 3D model (also referred as target) presented in the model viewer is used for this analysis and the only values necessary to SMART as input are the model altitude and the distance between the target and the sensor. Figure 63 shows an example where a sun irradiance spectrum is generated.

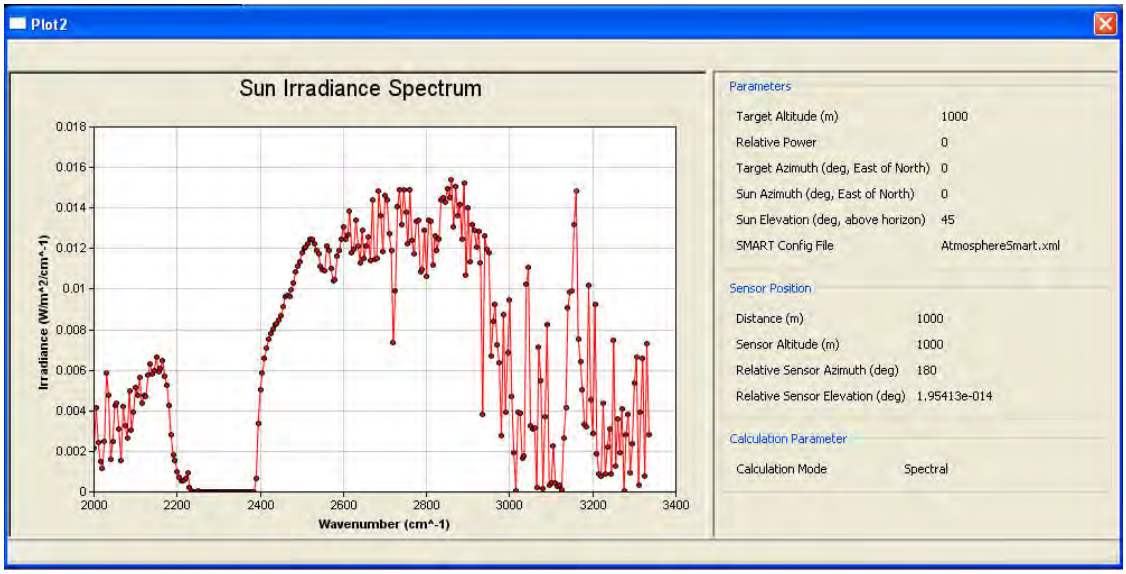

Figure 63: Sun irradiance spectrum obtained from SMART.

11 Using OSG formats within the IRSG

The IRSG uses OpenFlight models because this format contains `IRColor` and `IRMaterial` fields which are used by the IRSG to compute the thermal radiance of 3D models. These attributes are native in OpenFlight models but are not in OSG files. A mechanism to convert OpenFlight models in OSG format is then required. This is a first step to have a format which will be modifiable by a custom editor, removing the dependency to OpenFlight format and its associated model editors.

An important requirement is to keep the OSG files compliant with its own format and readable on other computers, even if they do not have the appropriate (modified) reader to process the newly included parameters. In order to respect this criterion, the `UserData` field of OSG files was used.

11.1 Using the OSG *UserData* field

Any object inheriting from `osg::Object` and placed in the `UserData` attribute of a `Node` will automatically be written into the `UserData` field of that object inside an `.osg` file (c.f. `Object_writeLocalData` inside the OSG plugin).

However, these data structures will not automatically be written or read inside the binary format version of OSG. As revealed by the code (c.f. `DataOutputStream::writeObject()` and `DataInputStream::readObject()` methods of the IVE plugin), only objects from the following classes (or inheriting from) can be read/write from/to an `.ive` file from/to the `UserData` field without modifying the plug-in:

- `osg::Node`;
- `osg::StateSet`;
- `osg::StateAttribute`;
- `osg::Drawable`; and
- `osgSim::ShapeAttributeList`.

11.2 Required modifications

The `ShapeAttributeList` data structure perfectly suits the needs of the IRSG. As stated before, this structure is automatically managed for the `UserData` field. No modifications to the OSG and IVE plug-ins are necessary to read or write the IRSG parameters. The `ShapeAttributeList` data structure is a container of `ShapeAttribute`. A `ShapeAttribute` is an object defined by:

- a name (`const char *`); and
- a type value (`int`, `double`, `string`).

So, the IRColor and IRMaterial OpenFlight fields are used as names for a ShapeAttribute and their value (database indices) are stored as integers. However, those fields are not supported in OSG nodes. The file GeometryRecords.cpp has been modified to read those fields and store them as UserData. Thus, the osgdb_openflight.dll is modified to include the IRColor and IRMaterial parameters in the OSG structure.

In flt::Face::readRecord() method, the following code is used to gather IRColor and IRMaterial in the geometries:

```
// Set IRColor and IRMaterial properties (not available in OSG)

osgSim::ShapeAttributeList* attributeList = new osgSim::ShapeAttributeList();

osgSim::ShapeAttribute* irColorAttribute = new osgSim::ShapeAttribute("IRColor", IRColor);

osgSim::ShapeAttribute* irMaterialAttribute = new osgSim::ShapeAttribute("IRMaterial",
IRMaterial);

attributeList->push_back(*irColorAttribute);
attributeList->push_back(*irMaterialAttribute);

_geometry.get()->setUserData(attributeList);
```

11.3 Converting a model (FLT to OSG)

A simple way to convert a model from a format to another one is to create a batch file in the same folder as the model to be converted. This batch file must invoke the osgconv.exe utility located in %KARMA_ROOT%\Softwares\OpenSceneGraph\bin\.

In the example presented in Table 23, osgconv is converting the CC130_IR model from an OpenFlight format to an OSG binary format. Notice that the PATH is also defined to ensure that the correct dll (osgdb_openflight.dll), modified from the original one, is loaded.

Table 23: An example of batch file used to convert a 3D model from FLT to IVE.

```
@echo off
PATH=%KARMA_ROOT%\Softwares\OpenSceneGraph\dll
"%KARMA_ROOT%\Softwares\OpenSceneGraph\bin\osgconv.exe" -O “preserveObjectpreserveFace” CC130_IR.flt
CC130_IR.ive
pause
```

12 Scaling parameters

To perform Monte Carlo based simulations, a requirement emerged stating that surface' temperature and emissivity can vary dynamically. These values are defined in a binary file (database) which is associated to a model. Instead of manipulating directly the binary file, the solution developed uses two new files (two files are required for each database to be modified): one containing the temperatures which must vary and the second, containing emissivity parameters to change.

Thus, for each parameter that needs to vary, three parameters must be defined:

1. minimal limit;
2. maximal limit; and
3. distribution type:
    a. Uniform Distribution : `0`
    b. Normal Distribution : `1`
    c. Exponential Distribution : `2`
    d. Laplace Distribution : `3`
    e. Cauchy Distribution : `4`
    f. Rayleigh Distribution : `5`
    g. Log Normal Distribution : `6`
    h. Levy Alpha Stable Distribution : `7`
    i. Gamma Distribution : `8`
    j. Chi-Squared Distribution : `9`
    k. F Distribution : `10`
    l. T Distribution : `11`
    m. Beta Distribution : `12`
    n. Pareto Distribution : `13`
    o. Poisson Distribution : `14`

A KARMA internal library (`%KARMA_ROOT%\Utilities\MonteCarlo`) is used to obtain a random value from these parameters which is then applied to a material's temperature and/or emissivity. Notice that the random values are obtained when the database is loaded i.e. when a 3D model is added in the scene generation module.

To control the value, `min` and `max` can be set to the same value. For example, if `min` and `max` are set to `1`, the value returned by the distribution will be `1`.

Table 24: An example defining a scale parameter inside an XML file.

```
<data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Karma\xml\schema\parameters.xsd">
<fileType>Parameters</fileType>
<parameter name="1"> <!-- database indice -->
        <vector3>
                <double>0.0</double> <!-- min -->
                <double>1.0</double> <!-- max -->
                <double>0</double> <!-- distribution -->
        </vector3>
<documentation>flat gray paint</documentation>
</parameter>
</data>
```

At run time, the scales are loaded from XML files and stored into the database that is associated to a 3D model. This database is used by the IRSG to compute the apparent radiance as presented in Section 4. Note that a temperature scale is not used when a temperature is based on a user defined spectrum (introduced in Section 8.1) while a emissivity scale is always used, no matter if the surface's emitted radiance is computed from a temperature or a user defined spectrum.

For KARMA simulations, scales are automatically loaded and used if they respect the following rules:

- The names of files must be the same as the 3D model files with one of the following the concatenated string, depending on the scale type, `(emissivitiesScalings).xml` or `(temperaturesScalings).xml`. For example, for the model `CC130_IR.flt`, the name shall be:
    - `CC130_IR(emissivitiesScalings).xml`, or
    - `CC130_IR(temperaturesScalings).xml`
- The files must be located in the same folder as the 3D model.

For SMAT analysis, it shall be explicitly set if scales must be used. This is done in the options dialog box as shown in Figure 64.

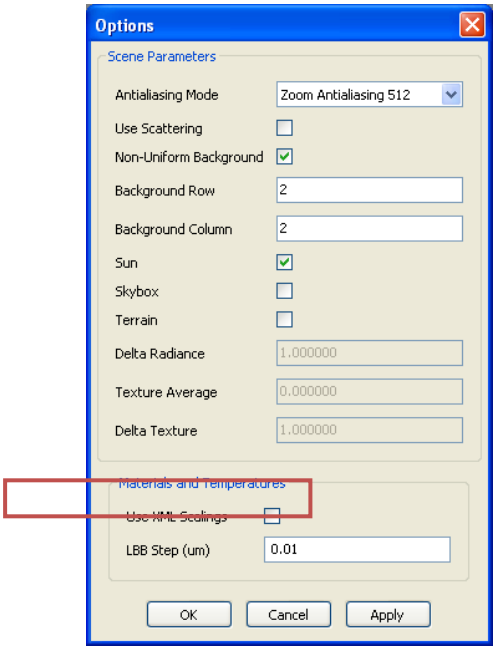

Figure 64: Setting the scales activation inside SMAT.

Notice that SMAT can create automatically the two scale files. When a user edits a temperature and material database, and this database is saved on disk via the *Save Database* command in the *File* menu, the files are generated (if they do not exist!) with default values.

13 Evaluating Performance Validator tool

The IRSG consumes a lot of computer resources and its performance affect the duration of KARMA simulations. In order to find bottlenecks, a commercial tool was used to evaluate the time required to perform the scene generation process. Performance Validator is a commercial tool which allows profiling code execution. Many statistics are available like the total time a method took to execute, the number of times a method was called, etc. An important aspect of Performance Validator is that it is not intrusive: the tool binds itself to binary files: no tags or additional code need to be inserted in the code.

Prior using this tool, the accuracy of timing results obtained with Performance Validator was evaluated. To accomplish this task, a simple in-house tool (name *Custom Tool* in the remainder of this section) using standard timing functions inserted directly in the source code to be evaluated was developed. This tool allows obtaining the total timing for different methods i.e. the cumulative method duration during for the whole software lifetime.

The following sections present the results obtained with Performance Validator and Custom Tool to calculate some code execution duration. These are just some basic tests to obtain a certain level of confidence with the tool.

13.1 Evaluating Performance Validator overhead

There are different performance timing mechanisms available in Performance Validator. As shown in Figure 65, *Performance Counters* mechanism was used to realize the timing operations. This is the most accurate method available.

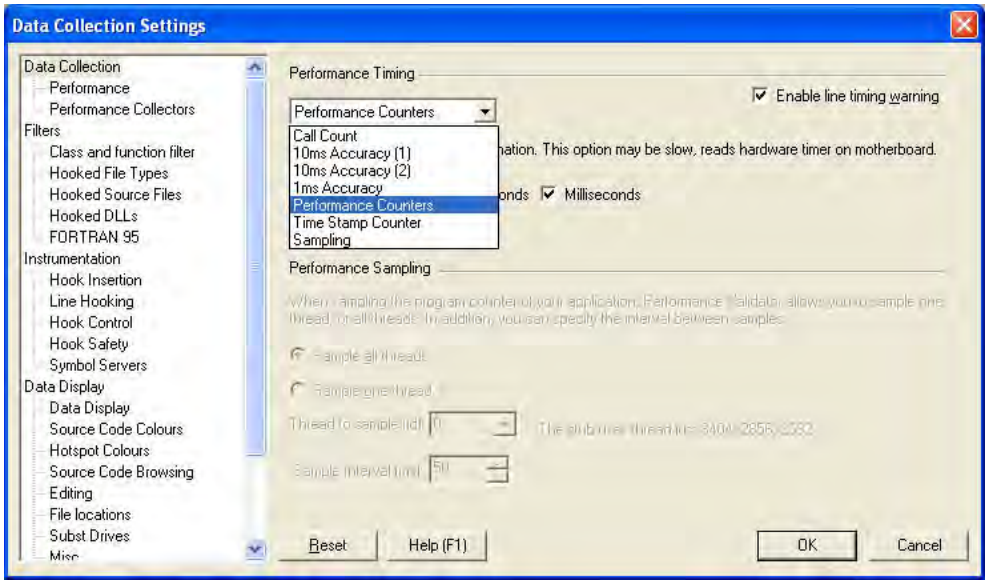

Figure 65: Performance Validator timing mechanisms.

13.2 Test 1: Evaluating method without child calls

An important concern was that Performance Validator is adding a lot of code to hook/analyze a software component. The result would then be that extra time, i.e. overhead, is added to method duration. To analyze this phenomenon, a method with an *a priori* estimated time was defined. By using the C++ `Sleep(DWORD dwMilliseconds)` function, which pause the normal execution flow in terms of millisecond (specified in argument), the theoretic time to execute a portion of code is known before its execution. Then, it is possible to observe a tool precision to profile this method.

For different reasons mentioned in the Performance Validator user guide, some methods may not be instrumented by the profiler (the method is too short for instance). If the method to be tested only contains a single call to `Sleep(…)`, it will not be evaluated. To be profiled, the method shall contain extra code (some calls to `std::cout()`or `Sleep(0)` for instance).

Table 25: Code for the evaluation of a method call.

```
int main(int argc, char* argv[])
{
    Run();
}

void Run()
{
     Sleep(10000);// 10,000 ms sleep
}
```

Table 26 presents the results obtained for the preceding code (i.e. profiling the `Run()` method) with Performance Validator and the tool developed for comparison purpose.

Table 26: Evaluation of a portion of code results.

Profiling Tool	Duration (ms)
Performance Validator	9,999.63
Custom Tool	10,001.03

Thus, Performance Validator does not add significant extra time to calculate a method which does not include calls to other methods.

13.3 Test 2: Evaluating a method with repeated child calls

Another concern was that Performance Validator is adding a lot of code to hook/analyze a method which calls other methods, adding an extra time to method duration.

Table 27: Code for the evaluation of a method call with 10,000 child calls.

```
int main(int argc, char* argv[])
{
    Run();
}

void Run()
{
     for(int i = 0; i < 10000; i++)
     {
         ShortSleep();
     }
}

void ShortSleep()
{
    Sleep(1); // 1 ms sleep
}
```

Table 28 presents the results obtained for the previous code (profiling the `Run()` and the `ShortSleep()` methods) with Performance Validator and the custom tool developed.

Table 28: Repeated method calls results.

Profiling Tool	Total time (ms)	
	Run	**ShortSleep**
Performance Validator	19,538.27	19,483.61
Custom Tool	19,538.99	19,522.21

Notice that 10,000 calls to a method containing a 1 ms sleep take more than 10,000 ms to execute since there is an extra cost to call a non-inline method, independently from its content. With the results obtained, Performance Validator gives accurate results for a method which call a repeated (but relatively low compared to the next test) number of child methods.

13.4 Test 3: Evaluating a method with numerous child calls

Another test was elaborated to observe a method which calls another method very often. With preliminary tests done with Performance Validator, the tool seems to overestimate some method duration when these methods are called a high number of times.

Table 29: Code for the evaluation of a method call with 1,000,000 child calls.

```
std::vector<int> values;

int main(int argc, char* argv[])
{
    Run();
}

void Run()
{
    for(int i = 0; i < 1000000; i++)// 1,000,000 calls
        IncreaseVector();
}

void IncreaseVector()
{
    values.push_back(1);
}
```

Table 30 presents the results obtained for the previous code (profiling the `Run()` and the `IncreaseVector()` methods) with Performance Validator and the custom tool developed.

Table 30: Numerous method calls results.

Profiling Tool	Total time (ms)	
	Run	**IncreaseVector**
Performance Validator	12,175.60	9,979.53
Custom Tool	9,943.97	9,227.98

13.5 Discussion

Performance Validator has proved to be an interesting tool helping in the process of obtaining C++ methods execution time. This software is easy to use and do not required code modifications in classes to be profiled. The tested case consisting to obtain the execution times required for a method with numerous child calls showed that there could be situations where the tool is not totally accurate. However, there are no equivalent situations in the IRSG module.

14 Conclusion

During the contract “Synthetic Infrared Scene” (W7701-082234), many new features were developed to increase the fidelity of the infrared scene generator module of the KARMA framework. The signature and modelling analysis tool (SMAT) was also modified to take advantage of the recent development. New tools were also added to SMAT to support the analysis.

The main improvements to the IRSG module include: the use of advanced rendering libraries and mechanisms to exploit graphical processor units, better atmospheric modelling including the use of a wideband-ck mode for increased performances, better representation of backgrounds, better representation of surface reflections, implementation of a zoom antialiasing algorithm, and representation of scattering effects.

At this moment, the zoom antialiasing algorithm is working fine but some optimizations are necessary to decrease the time require to generate an image. In fact, the performances of the IRSG still need to be addressed because it has a major impact on the simulations. To this end, the tool Performance Validator could be used to detect bottlenecks and make necessary changes and optimizations.

As presented in Section 9.3, the scattering effect is not fully functional. Indeed, the process generates a result which looks like a cross instead of a diffuse circle. Some investigations should allow detecting what stage of the mechanism produces this unexpected result.

References

[1] Lepage, J., Labrie, M., Rouleau, E., Richard, J., Ross, V., Dion, D., Harrison, N., "DRDC's approach to IR scene generation for IRCM simulation" in Technologies for Synthetic Environments: Hardware-in-the-Loop XVI, edited by Scott B. Mobley, Proceedings of SPIE Vol. 8015 (SPIE, Bellingham, WA 2011) 80150F.

[2] Rouleau, E. (2008). “Infrared Scene Generation (IRSG): Developer’s Guide”, DRDC-Valcartier CR 2008-258.

[3] Richard, J. (2008). “Signature Modelling and Analysis Tool (SMAT): User’s Guide”, DRDC-Valcartier CR 2008-260.

[4] Richard, J. (2008). “Signature Modelling and Analysis Tool (SMAT): Developer’s Guide”, DRDC-Valcartier CR 2008-259.

[5] Lepage, J.-F., Rouleau, E., Richard, J., & Harrison, N. (2010). “Infrared scene generation for countermeasures simulations: Implementation in the KARMA framework, phase 1”, DRDC Valcartier TR 2010-284, in publication process, UNCLASSIFIED.

[6] Ross, V. (2010). The SMARTI library.

[7] Ross, V. and Dion, D., " SMART and SMARTI: visible and IR atmospheric radiative transfer libraries optimized for wide-band applications," Proc. SPIE 8014, paper 26 (2011).

[8] Sills, T. G., & Williams, O. M. (2004). Aliasing and scintillation reduction in real-time computer graphics. Optical Engineering, 43(8), 1908-1915.

[9] Sills, T. G. (June 2006). Anti-aliasing for infrared scene generation using programmable graphics and the NVIDIA Cg toolkit. Optical Engineering, 066401-1,066401-5.

[10] Mountain. (2011). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Mountain.

[11] Tremblay, G., Roy, G., & Cao, X. (2010, December). Imaging through aerosols: a simple model based on the Modulation Transfer Function properties. RDDC-Valcartier.

Annex A AtmosphereSmart XML parameters file example.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!--Created by KARMA XML file logger-->
<!--
Copyright Her Majesty the Queen as represented by the Minister of National Defence, 2011

Terms of release:
The information contained herein is proprietary to Her
Majesty and is provided to the recipient on the
understanding that it will be used for information and
evaluation purposes only. Any commercial use
including for manufacture is prohibited. Release to
third parties of this publication or information
contained herein is prohibited without the prior
written consent of Defence R&D Canada.
-->
<data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="..\..\..\..\..\..\xml\schema\parameters.xsd">
  <fileType>Parameters</fileType>
  <type>/AtmosphereSmart/Root/Atmosphere/AtmosphereSmart</type>
  <Object_Model_Identification_Table>
    <Name/>
    <Version/>
    <Date/>
    <Purpose/>
    <Application_Domain/>
    <Sponsor/>
    <POC/>
    <POC_Organization/>
    <POC_Telephone/>
    <POC_Email/>
  </Object_Model_Identification_Table>

  <documentation>Atmospheric model based on SMART/MODTRAN allowing the calculation of
transmittance, path radiance, background radiance, solar irradiance, up flux, down flux;
in spectral and/or wideband correlated-k.</documentation>

  <parameter name="AerosolModel">
    <double>0</double>
    <documentation>Aerosol model.

                        0: MODTRAN rural aerosol model
                        1: MODTRAN urban aerosol model
                        2: MODTRAN maritime aerosol model
                        3: MODTRAN nam aerosol model
                        4: MODTRAN tropospheric aerosol model
                        5: MODTRAN advective fog aerosol model
                        6: MODTRAN radiative fog aerosol model
                        7: MODTRAN desert aerosol model
        </documentation>
  </parameter>

  <parameter name="AirMass">
    <double>3</double>
    <documentation>Air mass parameter, ICSTL parameter in MODTRAN.</documentation>
  </parameter>

  <parameter name="AlbedoValues">
    <vector>
      <double>0.71</double>
      <double>0.56</double>
```

```
    <double>0.38</double>
    <double>0.13</double>
    <double>0.2</double>
    <double>0.2</double>
    <double>0.2</double>
    <double>0.18</double>
    <double>0.15</double>
    <double>0.12</double>
    <double>0.1</double>
    <double>0.08</double>
  </vector>
  <documentation>Surface albedo (unitless).</documentation>
</parameter>

<parameter name="AlbedoWavelengths">
  <vector>
    <double>1.5</double>
    <double>2</double>
    <double>2.5</double>
    <double>3</double>
    <double>3.5</double>
    <double>4</double>
    <double>5</double>
    <double>6</double>
    <double>8</double>
    <double>10</double>
    <double>12</double>
    <double>14</double>
  </vector>
  <documentation>Spectral grid of albedo data (microns).</documentation>
</parameter>

<parameter name="AtmosphereModel">
  <double>2</double>
  <documentation>Selects one of the six geographical-seasonal model atmospheres.

                      1: Tropical Atmosphere (15 deg North Latitude).
                      2: Mid-Latitude Summer (45 deg North Latitude).
                      3: Mid-Latitude Winter (45 deg North Latitude).
                      4: Sub-Arctic Summer (60 deg North Latitude).
                      5: Sub-Arctic Winter (60 deg North Latitude).
                      6: 1976 US Standard Atmosphere.
      </documentation>
</parameter>

<parameter name="BaseWavelengthMax">
  <double>5</double>
  <documentation>Upper spectral boundary (max. is 40 micron).</documentation>
</parameter>

<parameter name="BaseWavelengthMin">
  <double>3</double>
  <documentation>Lower spectral boundary (min. is 0.2 micron).</documentation>
</parameter>

<parameter name="CloudBaseAltitude">
  <double>-1</double>
  <documentation>Cloud base altitude relative to ground level (CALT parameter in
                MODTRAN)

                      greater than or equal to 0 : Cloud base altitude relative to
                                                   ground level (m).
                      less than 0 : Use default cloud base altitude.
       </documentation>
</parameter>

<parameter name="CloudExtinction">
```

```
  <double>0</double>
  <documentation>Cloud liquid water droplet and ice particle vertical extinction (CEXT
                 parameter in MODTRAN)

                       greater than 0 : Cloud water particle vertical extinction (m-1).
                       less than or equal to 0 : Do not scale extinction coefficients.
       </documentation>
</parameter>

<parameter name="CloudModel">
  <double>0</double>
  <documentation>MODTRAN cloud/rain model (0-10), ICLD parameter in MODTRAN.

                      0: No clouds or rain.
                      1: Cumulus cloud layer: base 0.66 km, top 3.0 km.
                      2: Altostratus cloud layer: base 2.4 km, top 3.0 km.
                      3: Stratus cloud layer: base 0.33 km, top 1.0 km.
                      4: Stratus/stratocumulus layer: base 0.66 km, top 2.0 km.
                      5: Nimbostratus cloud layer: base 0.16 km, top 0.66 km.
                      6: 2.0 mm/hr ground Drizzle (modeled with cloud 3 and 0.86 mm / hr
                         at 1.0 km).
                      7: 5.0 mm/hr ground Light rain (modeled with cloud 5 and 2.6 mm /
                         hr at 0.66 km).
                      8: 12.5 mm/hr ground Moderate rain (modeled with cloud 5 and 6.0
                         mm / hr at 0.66 km).
                      9: 25.0 mm/hr ground Heavy rain (modeled with cloud 1 and to 0.2
                         mm / hr at 3.0 km).
                      10: 75.0 mm/hr ground Extreme rain (modeled with cloud 1 and 1.0
                         mm / hr at 3.0 km).
                      18: Standard Cirrus model (64 mm mode radius for ice particles).
                      19: Sub-visual Cirrus model (4 mm mode radius for ice particles).
      </documentation>
</parameter>

<parameter name="CloudThickness">
  <double>0</double>
  <documentation>Cloud thickness (CTHIK parameter in MODTRAN)

                       greater than 0 : Cloud vertical thickness (m).
                       less than or equal to 0 : Use default cloud thickness.
       </documentation>
</parameter>

<parameter name="GroundTemperature">
  <double>20</double>
  <documentation>Lower surface (ground or sea) temperature (C).</documentation>
</parameter>

<parameter name="IrradianceMode">
  <double>1</double>
  <documentation>Irradiance mode.

                      1: 1D pre-calculation mode
                      2: 2D on-the-fly calculation mode
      </documentation>
</parameter>

<parameter name="MeasurementHeight">
  <double>5.0</double>
  <documentation>Measurement height for temperature, pressure and relative humidity (m).
                 Valid from 2 to 40 m. Only affects DRDC meteo model.
  </documentation>
</parameter>

<parameter name="MeteoModel">
  <double>1</double>
  <documentation>Use standard MODTRAN models, or modified MODTRAN models (temperature,
```

```
                  pressure and water content profile of the lower atmosphere).

                      1: Modtran model
                      2: DRDC model
       </documentation>
</parameter>

<parameter name="ModelType">
  <entityType>SMART_ATMOSPHERIC_MODEL</entityType>
  <documentation>Used to save the model type.
  </documentation>
</parameter>

<parameter name="PressureMH">
  <double>1013</double>
  <documentation>Pressure at measurement height (mbar). Valid from 800 to 1200 mbar.
                Only affects DRDC meteo model.
  </documentation>
</parameter>

<parameter name="RelativeHumidityMH">
  <double>50</double>
  <documentation>Relative humidity at measurement height (%). Valid from 0 to 100 %.
                Only affects DRDC meteo model.
  </documentation>
</parameter>

<parameter name="Resolution">
  <double>5</double>
  <documentation>Wavenumber resolution.

                      1:  1 cm-1
                      5:  5 cm-1
                      15: 15 cm-1
      </documentation>
</parameter>

<parameter name="ScatteringMode">
  <double>2</double>
  <documentation>Scattering approximation mode affects the accuracy (and speed) of the
                scattering calculations. Setting the scattering mode to single
                scattering is faster but less accurate.

                      1: single scattering
                      2: two flux multiple scattering (required for clouds modelisation
                         and up-down flux calculation)
      </documentation>
</parameter>

<parameter name="TemperatureALT0">
  <double>20</double>
  <documentation>Air temperature (altitude 0) (C). Valid from -10 to 40 C. (the absolute
                difference between this parameter and TemperatureMH cannot exceed 10
                C.). Only affects DRDC meteo model.</documentation>
</parameter>

<parameter name="TemperatureMH">
  <double>20</double>
  <documentation>Air temperature at measurement height (C). Valid from -40 to 40 C. (the
                absolute difference between this parameter and TemperatureALT0 cannot
                exceed 10 C.). Only affects DRDC meteo model.</documentation>
</parameter>

<parameter name="Visibility">
  <double>0</double>
  <documentation>Koschmieder visibility (m), 0 for model default.</documentation>
</parameter>
```

```
  <parameter name="WindDirection">
    <double>0</double>
    <documentation>Current wind direction (East of North) (rad).</documentation>
  </parameter>

  <parameter name="WindSpeed">
    <double>5</double>
    <documentation>Wind speed at measurement height (m/s). Valid from 0.1 to 30 m/s.
    </documentation>
  </parameter>

  <parameter name="WindSpeedAverage24">
    <double>5</double>
    <documentation>Wind speed average in last 24 hours (m/s). Valid from 0.1 to 30 m/s.
    </documentation>
  </parameter>

  <parameter name="WindSpeedMeasurementHeight">
    <double>10</double>
    <documentation>Measurement height for wind speed (m). Valid from 2 to 40 m.
    </documentation>
  </parameter>
</data>
```

List of symbols/abbreviations/acronyms/initialisms

2D	Two Dimensions
3D	Three Dimensions
API	Application Program Interface
ATM	Atmosphere
CG	C for Graphics
CK	Correlated-K
CPU	Central Processing Unit
CR	Contract Report
CSV	Comma-Separated Values
DND	Department of National Defence
DRDC	Defence Research & Development Canada
FBO	Framebuffer Object
FFT	Fast Fourier Transform
FOI	Field Of Interest
FOV	Field Of View
FSAA	Full Screen Antialiasing
GPL	General Public License
GPU	Graphics Processing Unit
HDR	High Dynamic Range
HUD	Heads-Up Display
ICD	Installable Client Driver
ID	Identifier
IR	Infrared
IRSG	Infrared Scene Generator
LBB	Black Body Radiance
LOD	Level Of Details
LOS	Line Of Sight
MTF	Modulation Transfer Function
OpenGL	Open Graphic Library
OS	Operating System

OSG	OpenSceneGraph
PSF	Point Spread Function
R&D	Research & Development
RTT	Render To Texture
SMAT	Signature Modelling and Analysis Tool
XML	Extensible Markup Language
ZAA	Zoom Antialiasing

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR(The name and address of the organization preparing the document.Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Louis Tanguay Informatique inc.
825 Boul. Lebourgneuf, Bureau 204
Québec, Canada
G2J 0B9

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE(The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Synthetic Infrared Scene: Improving the KARMA IRSG module and signature modelling tool SMAT

4. AUTHORS(last name, followed by initials – ranks, titles, etc. not to be used)

Labrie, M.-A.; Rouleau E.; Richard J.; Bastien A.; Desmeules M.; Rivest-Sabourin G.

5. DATE OF PUBLICATION (Month and year of publication of document.)

March2011

6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)

106

6b. NO. OF REFS (Total cited in document.)

11

7. DESCRIPTIVE NOTES(The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY(The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO.(If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

Project 13ng

9b. CONTRACT NO.(If appropriate, the applicable number under which the document was written.)

W7701-082234/001/QCL

10a. ORIGINATOR'S DOCUMENT NUMBER(The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

LTI-SIS-2011-1

10b. OTHER DOCUMENT NO(s).(Any other numbers which may be assigned this document either by the originator or by the sponsor.)

DRDC Valcartier CR 2011-167

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to theDocument Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT(A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The main objective of the contract “Synthetic Infrared Scene” (W7701-082234) was to increase the level of fidelity of infrared guided weapon engagement simulations inside the KARMA simulation environment. The work was carried out from November 2008 to March 2011. This contract report focuses on presenting the new functionalities that were added to the infrared scene generator (IRSG) module which is part of the KARMA framework. Modifications were also done to the signature modelling and analysis tool (SMAT) which uses the IRSG to perform various kind of analysis.

L'objectif principal du contrat **"Scène** Infrarouge Synthétique" (W7701-082234) a été d'augmenter le niveau de fidélité d’engagements impliquant des autodirecteurs infrarouges dans l'environnement de simulation KARMA. Le travail a été réalisé à partir de novembre 2008 jusqu’à mars 2011. Ce rapport de contrat est axé sur la présentation des nouvelles fonctionnalités qui ont été ajoutées au module de génération de scène infrarouge (IRSG) faisant partie de l’environnement KARMA. Des modifications ont également été apportées à l’outil de modélisation et d'analyse de signature infrarouge (SMAT) qui utilise l’IRSG pour effectuer différents types d'analyse.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Infrared scene generation; Infrared signature; Modelling and simulation (M&S); OpenGL; Shaders

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca